ESTRUTURAS, PESQUISA E ORDENAÇÃO DE DADOS

JEAN GOMES TURET
PIETRO MARTINS DE OLIVEIRA
ROGÉRIO DE LEON PEREIRA

ORGANIZADOR
JEAN GOMES TURET

ACESSE AQUI ESTE MATERIAL DIGITAL!

EXPEDIENTE

Coordenador(a) de Conteúdo
Nader Ghoddosi
Projeto Gráfico e Capa
Arthur Cantareli Silva
Editoração
Laura Janke
Design Educacional
Giovana Vieira Cardoso
Revisão Textual
Salen Nascimento Silva
Ilustração
Andre Luis Azevedo da Silva
Eduardo Aparecido Alves
Geison Ferreira da Silva
Fotos
Shutterstock e Envato

FICHA CATALOGRÁFICA

N964 Núcleo de Educação a Distância. **TURET,** Jean Gomes; **OLIVEIRA,** Pietro Martins de; **PEREIRA,** Rogério de Leon.

Estruturas, Pesquisa e Ordenação de Dados / Jean Gomes Turet; Pietro Martins de Oliveira; Rogério de Leon Pereira; organizador: Jean Gomes Turet. - Florianópolis, SC: Arqué, 2024.

176 p.

ISBN papel 978-65-6137-953-3
ISBN digital 978-65-6137-954-0

1. Estruturas 2. Pesquisa 3. Dados 4. EaD. I. Título.

CDD - 003

Bibliotecária: Leila Regina do Nascimento - CRB- 9/1722.

Ficha catalográfica elaborada de acordo com os dados fornecidos pelo(a) autor(a).

Impresso por:

AVALIE ESTE LIVRO!

ACESSE O QRCODE

CRIAR MOMENTOS DE APRENDIZAGENS INESQUECÍVEIS É O NOSSO OBJETIVO E POR ISSO, **GOSTARÍAMOS DE SABER COMO FOI SUA EXPERIÊNCIA.**

Conta para nós! leva **menos de 2 minutos** *Vamos lá?!*

DIGITE O CÓDIGO

02511793

RESPONDA A PESQUISA

RECURSOS DE IMERSÃO

PENSANDO JUNTOS

Este item corresponde a uma proposta de reflexão que pode ser apresentada por meio de uma frase, um trecho breve ou uma pergunta.

APROFUNDANDO

Utilizado para temas, assuntos ou conceitos avançados, levando ao aprofundamento do que está sendo trabalhado naquele momento do texto.

EU INDICO

Utilizado para agregar um conteúdo externo.

ZOOM NO CONHECIMENTO

Utilizado para desmistificar pontos que possam gerar confusão sobre o tema. Após o texto trazer a explicação, essa interlocução pode trazer pontos adicionais que contribuam para que o estudante não fique com dúvidas sobre o tema.

PRODUTOS AUDIOVISUAIS

Os elementos abaixo possuem recursos audiovisuais. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

PLAY NO CONHECIMENTO

Professores especialistas e convidados, ampliando as discussões sobre os temas por meio de fantásticos podcasts.

EM FOCO

Utilizado para aprofundar o conhecimento em conteúdos relevantes utilizando uma linguagem audiovisual.

INDICAÇÃO DE FILME

Uma dose extra de conhecimento é sempre bem-vinda. Aqui você terá indicações de filmes que se conectam com o tema do conteúdo.

INDICAÇÃO DE LIVRO

Uma dose extra de conhecimento é sempre bem-vinda. Aqui você terá indicações de livros que agregarão muito na sua vida profissional.

CAMINHOS DE APRENDIZAGEM

7

UNIDADE 1

OPERAÇÕES DE BUSCA 8

29

UNIDADE 2

ÁRVORES BINÁRIAS 30

OPERAÇÕES SOBRE ÁRVORES BINÁRIAS 50

71

UNIDADE 3

EXCLUSÃO EM ÁRVORES BINÁRIAS 72

ÁRVORES AVL 88

111

UNIDADE 4

ÁRVORES B 112

ALGORITMOS DE ORDENAÇÃO:
BUBBLE SORT, SELECTION SORT E INSERTION SORT 126

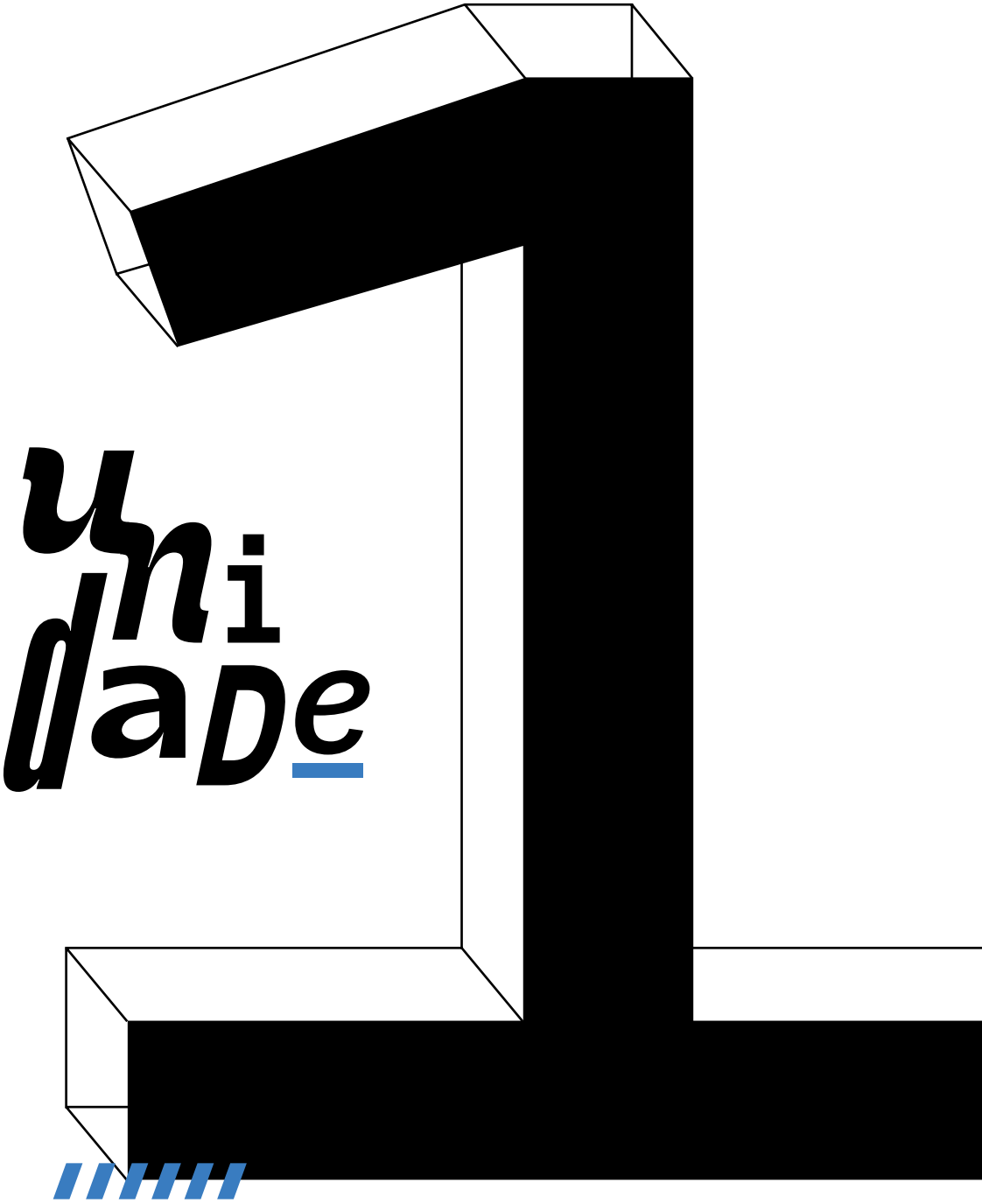
143

UNIDADE 5

ALGORITMOS DE ORDENAÇÃO AVANÇADOS 144

TABELA DE DISPERSÃO (*HASHING*) 158

Unicesumar



TEMA DE APRENDIZAGEM 1

OPERAÇÕES DE BUSCA

MINHAS METAS

- Compreender conceitos iniciais de dados.
- Entender a operação de busca sequencial.
- Compreender como funciona a estruturação de busca sequencial.
- Verificar as técnicas para desenvolvimento de busca sequência indexada.
- Aprender acerca dos algoritmos de estruturação de busca binária.
- Entender a busca por interpolação.
- Verificar, na prática, algoritmos de ordenação de buscas.

INICIE SUA JORNADA

Você e seus colegas estão trabalhando em um projeto para criar um aplicativo para dispositivos móveis que permitam a comunicação interna da escola. O School Connect é um programa projetado para ajudar alunos, professores e funcionários a se comunicarem melhor, permitindo que os usuários adicionem contatos, compartilhem informações e enviem mensagens.

O aplicativo funcionou bem no início. À medida que mais pessoas começaram a usá-lo, no entanto, a busca por contatos ficou cada vez mais lenta. Os usuários, agora, precisam esperar bastante tempo ou rolar por uma longa lista para encontrar um contato específico. A equipe de desenvolvimento tem medo de que isso possa prejudicar a experiência do usuário e diminuir o interesse dele no aplicativo.

Para garantir que a experiência do School Connect seja fluida e útil para os usuários, o próximo desafio é descobrir como melhorar a velocidade da busca e a estrutura dos dados.

Como você resolveria essas questões?

Algumas perguntas contribuem para o processo resolutivo da problemática, por exemplo:

1. Quais estruturas ou técnicas de dados podem ser usadas para otimizar a pesquisa no aplicativo de contatos?
2. Como esses métodos podem ser usados para garantir um bom desempenho em grandes quantidades de dados?

PLAY NO CONHECIMENTO

Neste episódio, discutiremos uma questão intrigante acerca da capacidade dos computadores de buscar informações em grandes volumes de dados. Já pensou como sites ou aplicativos rastreiam suas buscas tão rapidamente? Ou como pode ser quase instantaneamente possível buscar um item específico no inventário dos jogos? Isso é resultado de uma variedade de abordagens para a busca e organização de dados. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

VAMOS RECORDAR?

Vamos recordar a ordenação de buscas? Neste vídeo, exploraremos algumas das principais técnicas de ordenação e entenderemos como elas afetam a eficiência da busca em grandes volumes de dados.

Acesse: **https://docs.google.com/document/d/1jjONKxfudzUbuNVs3i3qvx6-TaoBjZY8dBcYg1l-gY8/edit**.

DESENVOLVA SEU POTENCIAL

Neste tema, estudaremos alguns métodos de pesquisa em grandes quantidades de dados em busca de uma determinada informação. Veremos como a organização dos dados torna o processo de busca mais eficiente. Como a operação de busca é uma tarefa muito comum na ciência da computação, o conhecimento desses métodos é de suma importância para todo o profissional que deseja se tornar um bom programador.

CONCEITOS BÁSICOS

Antes de nos aprofundarmos no estudo de algoritmos de busca, trataremos de alguns termos que utilizaremos neste tema.

As tabelas são uma estrutura comum em bancos de dados relacionais. Ele é formado por linhas e colunas. Cada linha da tabela serve como um registro único, enquanto as colunas servem como atributos ou campos do registro. Por exemplo, as colunas Nome, Endereço, Telefone e Email podem estar presentes em uma tabela, contendo informações acerca dos clientes, e cada linha é um registro que possui informações específicas do deles (Backes, 2023).

Já um arquivo é uma coleção de dados, que é armazenada em um sistema de arquivos, como um computador, os quais podem ser arquivos de texto, arquivos CSV (valores separados por vírgula), arquivos JSON e assim por diante. Eles normalmente incluem registros, entretanto, nem sempre estão organizados em um formato de tabela. Quando se trata de arquivos estruturados, como os CSV, eles ainda podem ser considerados uma série de registros.

Um registro, por sua vez, é uma única unidade de dados. Cada registro em uma tabela é uma linha inteira, que contém todas as informações relacionadas a um item ou entidade específica. É possível considerá-lo como um conjunto coerente de valores. Voltando ao exemplo da tabela de clientes, cada registro conterá informações de um cliente individual.

Normalmente, um registro é formado por campos ou atributos. Ele é composto por todos os campos, cada um representando um dado específico. Por exemplo, você pode ter campos como Nome, Idade e Endereço em um registro de cliente. O registro completo é formado pela combinação desses campos (Backes, 2023).

Dependendo da estrutura da tabela ou do arquivo, a complexidade dos registros pode variar. Eles podem ser simples, contendo apenas alguns detalhes básicos, ou complexos, contendo dados extremamente detalhados. Em um banco de dados relacional, um registro em uma tabela pode estar relacionado a outros registros.

Existe uma chave associada a cada registro. Se ela estiver dentro do próprio registro, é chamada de chave interna, chave embutida ou incorporada. No caso de a chave ficar fora do registro, ela é chamada de chave externa.

Para todos os registros dentro de uma tabela, pelo menos um dos campos precisa ter um valor único para, assim, diferenciar cada um dos registros. Esse campo é chamado de chave primária.

Por exemplo, imaginemos uma tabela de clientes. Cada registro dela representa um cliente e cada cliente possui um conjunto de dados, conforme a Tabela 1.

TABELA CLIENTES	
REGISTROS	CPF
	NOME
	RUA
	CEP
	TELEFONE

Tabela 1 – Exemplo de tabela de clientes e dados de um registro
Fonte: os autores.

Podemos identificar os clientes, de maneira única, pelo nome? Não, pois existem homônimos, ou seja, pessoas com o mesmo nome, como José Maria da Silva, Roberto Carlos dos Santos etc.

Também não podemos usar a Rua ou o CEP, pois vários clientes podem morar na mesma localidade. Se eu e minha esposa formos clientes dessa empresa, nosso Telefone também aparecerá mais de uma vez na tabela.

Para esse caso específico, sobrou o CPF (Cadastro de Pessoa Física), que é realmente único, uma vez que não existem duas pessoas, no Brasil, com o mesmo CPF. Se nenhum dos campos do registro fosse potencial para servir de chave primária, poderíamos criar um novo campo, chamado "codigo" ou "id", e atribuir um número único para cada um dos registros (Backes, 2023).

Ainda pensando nesse exemplo, imagine que a empresa dona dessas informações pretenda fazer uma ação de marketing e tenha como objetivo atingir apenas os clientes que residam numa determinada cidade, bairro ou rua. Podemos buscar, na Tabela Clientes, pelos campos Rua ou CEP. Quando um atributo com informações não únicas é utilizado numa pesquisa, ele é chamado de chave secundária.

Como você deve ter aprendido, as grandes massas de dados ficam armazenadas em dispositivos de memória secundária, como disco rígido, disco óptico, fita magnética, memória flash, dentre outros.

Todo dado, no entanto, para ser manipulado, precisa necessariamente estar na memória principal. Como a relação entre as duas memórias é desproporcional, ou seja, um computador tem Gb de memória RAM enquanto possui Tb de memória de armazenamento, somente uma pequena parcela dos dados pode ser analisada por vez.

Dessa forma, é muito mais inteligente carregar apenas uma tabela de chaves do que todos os registros de um arquivo. Encontrada a referência pretendida, busca-se no disco apenas a porção desejada de informação.

Alguns algoritmos são especializados em realizar buscas em arquivos de chaves primárias, outros em chaves secundárias. Alguns são mais rápidos em arquivos menores, outros ideais para grandes quantidades de informações. Um algoritmo de busca tem como objetivo, a partir de um argumento x recebido, encontrar numa tabela um registro que possua uma chave x equivalente. Ele retornará todos os dados do registro, ou o índice da sua posição num vetor, ou ainda um ponteiro para uma posição na memória.

Se não houver, no arquivo, um registro compatível com o argumento pesquisado, o algoritmo não obtém sucesso na sua busca e retorna uma mensagem de erro, um índice inválido ou um ponteiro nulo.

APROFUNDANDO

Um banco de dados é um sistema projetado para armazenar, administrar e recuperar dados de maneira eficaz. Desde pequenos negócios até grandes corporações e organizações governamentais, ele é vital em várias situações. Existem bancos de dados relacionais tradicionais e NoSQL atuais. Um banco de dados relacional usa a linguagem de consulta estruturada SQL para manipular os dados que são armazenados em tabelas compostas por linhas e colunas. Além disso, os bancos de dados NoSQL podem ter uma variedade de estruturas, como grafos, chaves-valor ou documentos, o que os torna mais adaptáveis ao lidar com dados não estruturados ou semiestruturados.

OPERAÇÃO DE BUSCA SEQUENCIAL

A busca sequencial é, de longe, a forma mais simples de pesquisa. Ela pode ser utilizada tanto para o caso em que a tabela está armazenada num vetor como numa lista ligada (Oliveira; Pereira, 2019).

A partir da primeira posição no vetor (ou do primeiro nó da lista), compara- se o valor atual da estrutura com o argumento x passado. Repete-se esse procedimento com cada um dos valores até que seja encontrado o valor desejado ou o final da tabela.

O Programa 1 traz um exemplo de busca sequencial em C. A função Busca Sequencial recebe três parâmetros: **o vetor (vec) a ser pesquisado, o argumento (arg) a ser procurado e o tamanho (tam) do vetor**.

São criadas duas variáveis locais. A variável "i" serve de índice para percorrer o vetor, enquanto a variável "achou" é usada para identificar se a busca foi concluída antes do final do vetor.

Como os vetores em C começam pela posição 0, a variável "i" também foi inicializada com esse valor para que a pesquisa realmente dê início pelo começo do vetor. A variável achou foi inicializada com -1.

Durante o laço de repetição é verificado se a variável de controle "i" atingiu o tamanho máximo "tam" recebido como parâmetro. O valor de "i" é incrementado sempre no final do laço, garantindo que a busca percorra o vetor até o final.

Adicionalmente, o laço possui outra regra de parada que verifica o valor da variável achou. Ela foi inicializada com -1, porém, se o valor for encontrado, a sua posição no vetor será atribuída ao valor de achou.

Ao final, a função retorna o valor da variável achou. Ele será -1 se a busca tiver falhado ou, no caso contrário, a posição do registro no vetor.

```
//Função de Busca Sequencial
int buscaSequencial(int vec[], int arg, int tam){
  int i = 0, achou = -1;
  while ((i < tam) && (achou == -1)){
    if (vec[i]==arg){
      achou = i;
    }
    i++;
  }
  return(achou);
}
```

Programa 1 – Busca sequencial / Fonte: os autores.

Descrição da Imagem: a figura retrata um algoritmo em C para uma função de busca sequencial. O algoritmo possui três estruturas principais. A primeira relacionada a um *while*; a segunda estrutura de condição (*if*); e a terceira estrutura relacionada a um contador i++. Fim da descrição.

Se considerarmos que os registros no vetor não estão organizados de forma alguma, a pesquisa poderá encontrar o valor desejado na primeira ou na última posição da estrutura. Imagine que a tabela seja muito grande e buscar alguma informação, procurando em todos os registros, pode ter um custo computacional muito elevado. Uma maneira de melhorar isso é a técnica de busca sequencial indexada.

BUSCA SEQUENCIAL INDEXADA

Não é possível prever qual será o valor buscado numa pesquisa e nem a posição em que o registro está armazenado no vetor, menos ainda se ele está ou não presente na tabela. Tudo depende do que está armazenado e o que se costuma pesquisar. Se, numa tabela de clientes, as buscas são feitas por pessoas que habitem na mesma região demográfica, para efetuar ações mais precisas e eficazes de marketing, uma boa ideia seria criar um índice específico ordenado por cidade, estado ou CEP (Oliveira; Pereira, 2019).

Se o síndico do prédio em que você mora entrega o boleto do condomínio, começando pelos apartamentos do último andar até o térreo, seria interessante que houvesse um índice com o número do apartamento ordenado de forma decrescente.

Em uma loja de departamentos especializada em produtos de moda feminina é muito provável que o usuário busque uma cliente do gênero feminino em vez de um homem, por isso, os dados devem ser ordenados, primeiramente, por gênero. Se for observado que alguns argumentos específicos são procurados com mais frequência, esses registros podem ser movidos para uma região mais próxima do início da tabela, fazendo com que futuras buscas sejam cada vez mais rápidas.

Há um método em particular, muito simples e que traz bons resultados, chamado método da transposição, no qual o registro recuperado com sucesso é trocado pelo registro imediatamente anterior. Imagine um vetor de valores inteiros que esteja ordenado de forma crescente. A busca sequencial procuraria o valor desejado em cada uma das posições até encontrá-lo ou até chegar ao final do arquivo, o que ocorrer primeiro. Mesmo no caso de o registro não existir na tabela, todo o arquivo será percorrido.

Entretanto, como sabemos que a estrutura está ordenada de forma ascendente, no momento em que for encontrado um número maior do que o do qual está sendo procurado, não há mais necessidade de continuar a pesquisa, porque certamente o valor procurado não estará no restante da tabela. Nessa linha de raciocínio, podemos dizer que compensa classificar um arquivo antes de pesquisar informações dentro dele.

APROFUNDANDO

Uma ordem crescente no vetor indica que cada elemento é menor ou igual ao próximo. Essa ordenação pode ser uma grande vantagem para otimizar o processo quando se busca um valor específico, como em buscas lineares ou sequenciais.

Se você encontrar um elemento que é maior do que o valor que você está procurando enquanto percorre um vetor ordenado em ordem crescente, é provável que você não precise continuar a busca. Isso ocorre, porque, como resultado do ordenamento do vetor, todos os elementos subsequentes também terão dimensões maiores. Como resultado, a busca pode ser interrompida imediatamente, o que economiza muito tempo encontrando o valor pretendido ou confirmando que ele não está no vetor.

A estratégia da busca sequencial indexada envolve a criação de uma tabela complementar ao arquivo de dados. Essa tabela é composta por pelo menos dois campos: a chave de busca e o endereço do registro correspondente no arquivo original. A ordenação da tabela de índice pode seguir diferentes critérios, como ascendente, descendente ou qualquer outro que melhor se adeque às principais operações de busca que farão uso da chave contida nessa tabela auxiliar.

Mais uma vez, mencionando uma suposta tabela de clientes como exemplo, fisicamente, os registros estão gravados em disco pela ordem como os clientes são cadastrados no sistema. Se a busca for efetuada pelo número do CPF do cliente, uma nova tabela é criada contendo o CPF e a posição original do registro na tabela de clientes.

A BUSCA BINÁRIA

A forma mais eficiente de efetuar pesquisa em um arquivo ordenado sem a necessidade de tabelas auxiliares é a busca binária. A estratégia consiste em comparar o argumento chave ao elemento do meio da tabela. Se forem iguais, a busca terá terminado com sucesso (Pereira, 2016).

No caso contrário, o vetor será divido em duas metades, e a pesquisa será repetida na metade inferior, se o argumento for menor do que o valor do meio da tabela, ou na parte superior, se o argumento for maior.

A cada iteração, a busca binária reduz a quantidade de possíveis candidatos pela metade. Vale ressaltar que essa estratégia funciona apenas em estruturas cujos dados se encontram ordenados.

Esse algoritmo não pode ser utilizado com listas dinâmicas devido às características da sua estrutura. Mesmo criando um ponteiro especial que aponte para o meio da lista, toda vez que o problema for dividido, ela precisará ser percorrida novamente para achar o meio de sua nova metade, o que torna o tempo de execução do algoritmo muito extenso.

Simulemos uma busca binária num vetor ordenado de dez posições, demonstrado na Figura 1. O valor do argumento a ser encontrado é 5.

A variável menor será inicializada com 0 e maior com 9 (tamanho – 1).

a) Para o cálculo do meio do vetor, fazemos o menor, a soma do menor valor e o maior valor e, com esse resultado, dividimos por dois. Assim, como a variável meio é do tipo inteiro, a posição que indica a metade do vetor é 4. O valor contido no vetor na posição meio é 16, que é maior do que o argumento pesquisado 5.

b) Como o valor não foi encontrado, o laço é repetido e a busca se dá entre as posições 0 a 3. Como a variável meio é do tipo inteira, a posição que indica a metade do vetor é 1. O valor contido no vetor na posição meio é 7, que é maior do que o argumento pesquisado 5.

c) Como o valor não foi encontrado, o laço é repetido e a busca se dá na parte restante do vetor, que é a posição 0. O vetor na posição 0 contém o valor procurado 5, o algoritmo finaliza com sucesso, recuperando a posição no vetor, cujo argumento está armazenado.

a)

0	1	2	3	4	5	6	7	8	9
5	7	8	12	15	20	21	30	31	33

b)

0	1	2	3	4	5	6	7	8	9
5	7	8	12	15	20	21	30	31	33

c)

0	1	2	3	4	5	6	7	8	9
5	7	8	12	15	20	21	30	31	33

d)

0	1	2	3	4	5	6	7	8	9
5	7	8	12	15	20	21	30	31	33

Figura 2 - Simulação de uma busca binária pelo elemento 5 num vetor de dez posições
Fonte: os autores.

Descrição da Imagem: na figura, há quatro tabelas principais relacionadas ao funcionamento de uma interação binária. O quadro possui dez números em ordem crescente, iniciando no 0 e indo até o valor 9. A partir das etapas binárias, verifica-se o alcance ao número desejado. Fim da descrição.

BUSCA POR INTERPOLAÇÃO

A busca por interpolação é outra forma de pesquisa em vetores com dados ordenados. Se os valores estiverem distribuídos de forma uniforme entre vec[menor] e vec[maior], esse método pode ser ainda mais eficiente do que a busca binária. Sua implementação é muito parecida com a busca binária, a qual acabamos de estudar.

A variável menor é definida com 0 e a maior é definida com tamanho – 1, sabendo que os valores estão distribuídos de forma uniforme e tomando por verdade que, em toda a execução do algoritmo, o argumento arg estará contido entre os valores vec[menor] e vec[maior] (Szwarcfiter; Markenzon, 2010).

A busca deve continuar até que o valor seja encontrado ou que menor > maior. Se arg for menor que vec[meio], o valor da variável maior será atualizado com meio -1, fazendo com que a busca aconteça na parte inferior do vetor. Caso contrário, a variável menor será atualizada com meio + 1 para que a busca prossiga na parte superior da estrutura.

Imaginemos a seguinte situação: você está em uma biblioteca e procura por um exemplar específico em uma prateleira em que os livros estão ordenados por título alfabeticamente. Uma maneira de começar a pesquisar um livro cujo título começa com a letra M é começar no meio da prateleira e encontrar a letra em que você está. Por exemplo, se estiver na letra G, você saberá que deve pular algumas partes para a frente, porque M vem depois da letra G. Você deve voltar para trás se estiver na letra T.

A busca por interpolação usa um conceito semelhante , mas tenta estimar como deve procurar com base no valor que está procurando em vez de ir direto para o meio. Em outras palavras, ela usa o valor desejado em relação ao início e ao fim da lista para tentar prever a posição provável do item na lista.

A busca por interpolação usa um conceito semelhante

Continuando com a analogia da biblioteca, é como se você tivesse uma noção aproximada do lugar em que, provavelmente, os livros com o nome M estão, dependendo de quão espalhados eles se encontram na prateleira. Você pode ir direto para perto da letra M se as letras estão distribuídas uniformemente. Se não forem, sua estimativa será um pouco mais imprecisa, mas ainda lhe dará uma boa indicação de como procurar.

Quando os dados têm uma distribuição previsível, essa técnica pode ser mais rápida que a busca binária, que começa no meio e reduz o problema pela metade. Entretanto, se a distribuição dos valores for desigual ou se houver muitos valores extremos que dificultam a tentativa de estimar o local certo, ela pode ser menos eficaz. A busca binária pode ser mais confiável nesses casos.

EM FOCO

Acesse seu ambiente virtual de aprendizagem e confira a aula referente a este tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

NOVOS DESAFIOS

O ordenamento de buscas em estruturas de dados é um tema central em ciência da computação, com aplicações que vão desde a organização eficiente de informações até a otimização de operações em bancos de dados e sistemas de recuperação de informações. Refletir acerca desse tema envolve considerar diversos aspectos, incluindo tipos de estruturas de dados, algoritmos de busca, critérios de ordenamento e a eficiência geral do sistema.

Para corrigir o problema de desempenho do aplicativo School Connect, visto no começo deste tema, a equipe deve estudar estruturas de dados e técnicas de busca que reduzam a pesquisa por contatos. As recomendações incluem:

- **Usar estruturas de dados otimizadas**: considere usar estruturas, como tabelas *hash* ou árvores binárias de busca, para acelerar a busca por nomes ou informações de contato.
- **Implementação de algoritmos de busca eficientes**: use algoritmos de pesquisa eficientes para reduzir o tempo de pesquisa e fornecer uma experiência de usuário mais rápida e responsiva.
- **Indexação e categorização de** dados: faça índices ou categorias para facilitar a busca de resultados rapidamente. A equipe também deve analisar a usabilidade do aplicativo e testar o desempenho para garantir que as melhorias realmente melhorem a experiência do usuário.

Ao concluir esses passos, a equipe pode garantir que o aplicativo School Connect seja fácil de usar e promova a comunicação e a cooperação entre alunos, professores e funcionários.

A natureza das diferentes estruturas de dados determina a maneira como as buscas podem ser realizadas e ordenadas. A busca sequencial é possível em estruturas lineares, como listas e *arrays*. Por outro lado, estruturas hierárquicas, como árvores, podem ser encontradas mais rapidamente por meio de algoritmos como a busca binária. Tabelas *hash* e outras estruturas permitem acesso rápido a elementos específicos, mas não sempre em uma ordem específica.

Um sistema deve ter um equilíbrio entre sua estrutura de dados, algoritmo de busca e critérios de ordenamento para ser eficiente. Considerar a dimensão do problema, a frequência das operações de busca e a necessidade de reorganizar os dados é uma boa prática. Essas decisões podem resultar em sistemas escaláveis e de alto desempenho.

Portanto, quando se pensa no ordenamento de buscas em estruturas de dados, deve-se pensar nas implicações práticas da escolha de estruturas e algoritmos, além das necessidades específicas do contexto em que são aplicados. Isso garante que o sistema atenda às necessidades de funcionalidade, escalabilidade e desempenho dos usuários.

VAMOS PRATICAR

1. No gerenciamento de bancos de dados relacionais, as tabelas desempenham um papel central na organização das informações. Cada tabela é composta por linhas e colunas, em que as linhas representam registros individuais e as colunas são campos ou atributos. Para um banco de dados de clientes, as colunas podem incluir informações como Nome, Endereço, Telefone e Email.

   Considerando uma tabela que contém informações acerca de clientes, qual das seguintes opções é um exemplo correto de um atributo ou campo que poderia estar presente nela?

   a) Uma lista de registros de compras de todos os clientes.
   b) Uma coluna com o nome do cliente.
   c) Uma linha contendo todas as informações de um cliente específico.
   d) Uma coluna contendo endereços de clientes únicos.
   e) Uma tabela que relaciona clientes com suas compras.

2. A busca sequencial é um método de pesquisa básico que pode ser aplicado tanto em vetores quanto em listas ligadas. O processo começa na primeira posição do vetor ou no primeiro nó da lista e compara o valor atual com o argumento x passado como parâmetro. Esse procedimento é repetido até que o valor desejado seja encontrado ou até que se alcance o final da estrutura.

   Qual das seguintes afirmações acerca da busca sequencial é verdadeira?

   a) A busca sequencial garante que o valor será encontrado antes de atingir o final da estrutura.
   b) A busca sequencial não pode ser usada para pesquisar em listas ligadas.
   c) A busca sequencial requer que os dados estejam ordenados para funcionar corretamente.
   d) A busca sequencial compara cada elemento da estrutura com o valor desejado até encontrá-lo ou alcançar o final da estrutura.
   e) A busca sequencial é o método mais rápido para encontrar um elemento em um vetor ou lista ligada.

3. A busca binária é uma estratégia eficiente para pesquisa em arquivos ordenados sem a necessidade de tabelas auxiliares. A técnica consiste em comparar o argumento-chave com o elemento do meio da tabela. Se forem iguais, a busca termina com sucesso. Se não, a tabela é dividida em duas metades, e a pesquisa continua na metade apropriada, conforme o valor da chave ser maior ou menor do que o elemento do meio.

   Com base nas informações apresentadas, avalie as asserções, a seguir, e a relação proposta entre elas:

   I - A busca binária é um método eficiente para encontrar um elemento em uma estrutura ordenada.

   PORQUE

   II - A busca binária divide a estrutura ao meio repetidamente, reduzindo o espaço de busca em cada iteração.

   É correto o que se afirma em:

   a) As asserções I e II são verdadeiras, e a II é uma justificativa correta da I.
   b) As asserções I e II são verdadeiras, mas a II não é uma justificativa correta da I.
   c) A asserção I é uma proposição verdadeira e a II é uma proposição falsa.
   d) A asserção I é uma proposição falsa e a II é uma proposição verdadeira.
   e) As asserções I e II são falsas.

REFERÊNCIAS

BACKES, A. R. **Algoritmos e estruturas de dados em linguagem C**. Rio de Janeiro: LTC, 2023.

OLIVEIRA, P. M. de, PEREIRA, R. de L. **Estrutura de dados II**. Maringá: Unicesumar, 2019.

PEREIRA, S. do L. **Estruturas de dados em C**: uma abordagem didática. São Paulo: Érica, 2016.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 2010.

CONFIRA SUAS RESPOSTAS

1. Alternativa B.

2. Alternativa D.

3. Alternativa A.

MEU ESPAÇO

Unicesumar

unidade

TEMA DE APRENDIZAGEM 2

ÁRVORES BINÁRIAS

MINHAS METAS

- Compreender conceitos iniciais de árvores.
- Entender a operação envolvida em árvores binárias.
- Compreender como funciona uma árvore estritamente binária.
- Entender conceitos fundamentais de árvores completas.
- Aprender acerca das árvores binárias diferentes.
- Compreender os passos envolvidos em árvores binárias diferentes.
- Verificar, na prática, as árvores binárias.

INICIE SUA JORNADA

A instituição educacional que você frequenta tem o objetivo de estabelecer um sistema que facilite a organização de arquivos de trabalhos, provas e materiais didáticos. Atualmente, todos os documentos estão armazenados em pastas físicas, mas isso tem causado alguns problemas, como perdas, dificuldade para encontrar materiais específicos e demora no tempo de busca.

A direção da escola tomou a decisão de digitalizar todos os documentos e estabelecer uma plataforma on-line para armazená-los. Entretanto, a equipe de TI tem dúvidas acerca de como organizá-los para facilitar a busca. Os arquivos podem ser organizados por data, tipo de documento ou disciplina, usando uma estrutura de árvore binária.

A responsabilidade dessa estrutura é sugerir uma solução baseada em árvores binárias, que facilite a organização e a busca de documentos. Considere como ordenar a árvore para facilitar a busca, lidar com a inserção de novos documentos e manter a estrutura fácil de se manter.

Caso deseje, escreva um parágrafo de cinco a dez linhas, identificando como você resolveria essa questão. Para melhorar a eficiência da operação da escola e dar aos alunos e professores uma experiência mais ágil na busca de materiais, é essencial resolver esse problema. Pode haver perda de tempo, frustração e, no pior dos casos, perda de documentos importantes se os documentos estiverem mal-organizados.

Algumas perguntas podem ajudar na resolução desse caso, por exemplo:

- **Quais regras devem ser seguidas para organizar os documentos na forma de uma árvore binária?**

Por exemplo: a data, o tipo de documento, a disciplina, o autor e outros fatores.

- **Como uma árvore binária pode ser construída para maximizar a inserção e a busca?**

Deve ser uma árvore binária de busca ou uma árvore balanceada.

- **Quais são as funções da árvore mais frequentes?**

Buscar, adicionar, remover, atualizar etc.

PLAY NO CONHECIMENTO

Você já ouviu falar de árvores que são binárias? Uma das estruturas mais importantes da ciência da computação, elas são usadas para buscar informações, organizar dados e até mesmo comprimir arquivos. Se você já se perguntou como os motores de busca encontram dados tão rapidamente ou como as árvores de decisão ajudam a inteligência artificial (IA), este episódio é para você. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

VAMOS RECORDAR?

Vamos recordar as árvores binárias? Neste vídeo, exploramos algumas das principais técnicas envolvidas e como elas afetam a eficiência da busca em grandes volumes de dados.

Acesse: **https://www.youtube.com/watch?v=dcOCsxiuKeU** .

DESENVOLVA SEU POTENCIAL

Neste tema, estudaremos uma solução útil para várias aplicações: a árvore binária. Definiremos algumas formas de estrutura de dados e mostraremos como elas podem ser implementadas na linguagem C. Como estrutura, a árvore é muito utilizada para organizar informações armazenadas tanto na memória principal como na secundária. Isso se dá devido ao fato de ser fácil e rápida a pesquisa de dados em árvores. Focaremos no entendimento da estrutura e não na sua definição matemática.

ÁRVORE BINÁRIA

Em ciência da computação e matemática, uma árvore binária é uma estrutura de dados que mostra uma hierarquia ou uma relação de ordem entre elementos. A árvore é chamada de binária, pois cada nó pode ter até dois filhos: um à esquerda e um à direita. Ela é composta por nós; cada nó contém uma chave ou valor e pode servir como referência para os filhos esquerdo e direito (Backes, 2023).

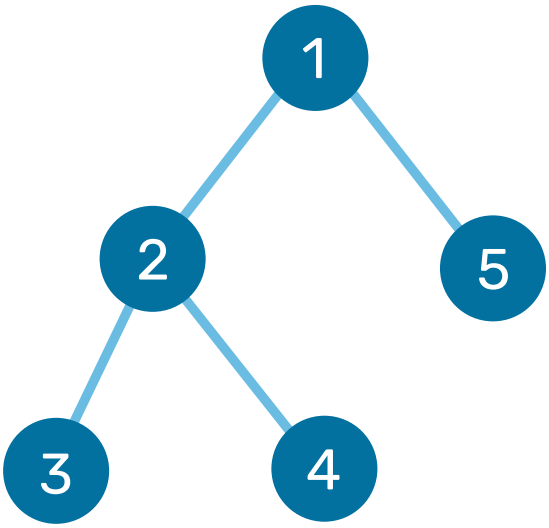


Figura 1 – Árvore binária
Fonte: https://pixabay.com/pt/vectors/estrutura-de-dados-%C3%A1rvore-bin%C3%A1ria-7390522/. Acesso em: 26 jul. 2024.

Descrição da Imagem: trata-se de uma figura em formato de nós e arestas, que representam uma árvore binária. A ilustração contém cinco nós interconectados uns com os outros. Fim da descrição.

A Figura 1 apresenta um exemplo de árvore binária, que possui nove nós, sendo A o nó raiz da árvore. Ela possui uma subárvore esquerda, com raiz em B, e uma direita, com raiz em C. A subárvore com raiz em B possui uma subárvore esquerda,

com raiz em D, e uma direita, com raiz em E. A subárvore com raiz em C possui uma subárvore esquerda vazia e uma subárvore direita, com raiz em F. As árvores binárias com raiz em D, G, H e I possuem subárvores direita e esquerda vazias.

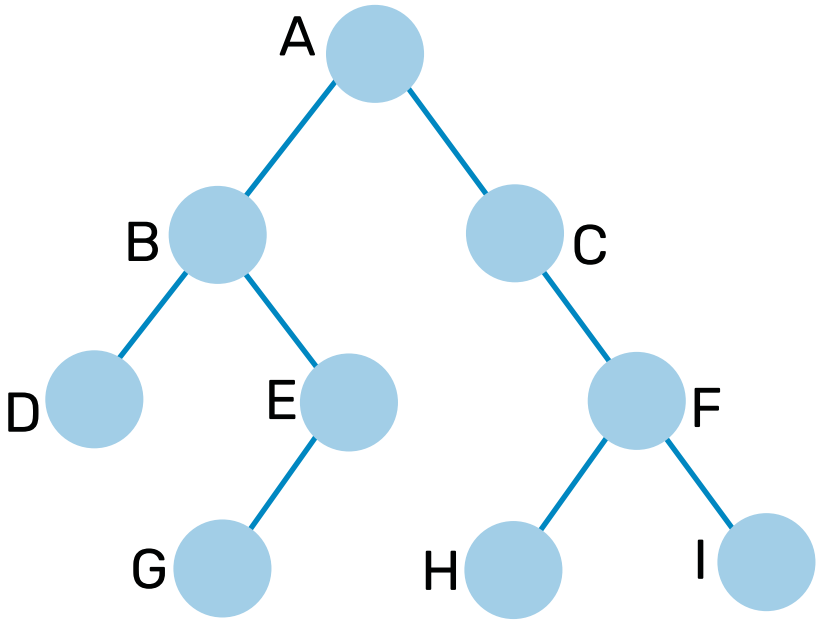


Figura 2 – Exemplo de árvore binária / Fonte: os autores.

Descrição da Imagem: exemplo de árvore binária com nove nós na cor azul interconectados por arestas (linhas). Fim da descrição.

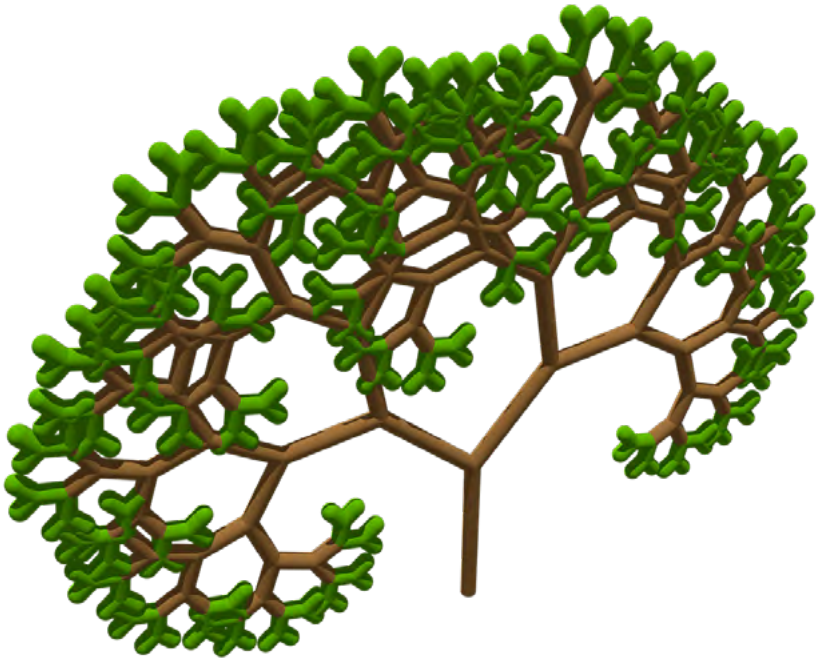
Se A é o nó raiz de uma árvore binária e tem uma subárvore com raiz em B, então, diz-se que A é pai de B e que B é filho à esquerda de A. De igual forma, se A tem uma subárvore direita com raiz em C, diz-se que A é pai de C e que C é filho à direita de A. Um nó sem filhos é chamado de folha. Na árvore representada pela Figura 1, os nós D, G, H e I são considerados folhas.

Os nós D e E são ambos filhos de B, dessa forma, podemos considerá-los nós irmãos. Também, podemos classificar os nós de acordo com a sua hereditariedade. O nó B é ancestral dos nós D, E e G, da mesma forma que esses três nós são descendentes de B.

Quando se percorre uma árvore a partir da raiz em direção às folhas, diz-se que se está caminhando para baixo ou descendo na árvore. De forma análoga, quando se está percorrendo uma árvore, a partir de uma de suas folhas em direção à raiz, diz-se que se está caminhando para cima ou subindo na árvore. Se fôssemos fazer uma analogia entre uma árvore na informática e uma árvore real (planta), teríamos a representação delas de cabeça para baixo na computação.

APROFUNDANDO

A seguir, demonstramos a estrutura fundamental de uma árvore binária:
Raiz: nó principal, da qual todos os outros nós derivam.
Filhos: pode-se esperar até dois filhos por nó; o nó pai geralmente tem filhos menores à esquerda do que à direita.
Folhas: aqueles de nós que não têm filhos.
Subárvores: são as partes de uma árvore que se ramificam de um nó.
Árvores binárias podem ser usadas em vários tipos de algoritmos, como busca, ordenação e expressões aritméticas. A organização dos nós em uma árvore binária torna as operações, como inserção, busca e exclusão de elementos, mais eficientes. Em comparação com listas ou outros tipos de estruturas de dados lineares, isso é um recurso mais eficaz.



ÁRVORE ESTRITAMENTE BINÁRIA

Uma árvore é considerada estritamente binária se todo nó que não for folha tiver sempre subárvores direita e esquerda não vazias. Na Figura 2, temos um exemplo de árvore estritamente binária. São considerados folhas os nós C, D, F e G. Os nós A, B e E possuem subárvores esquerda e direita não vazias (Oliveira; Pereira, 2019).

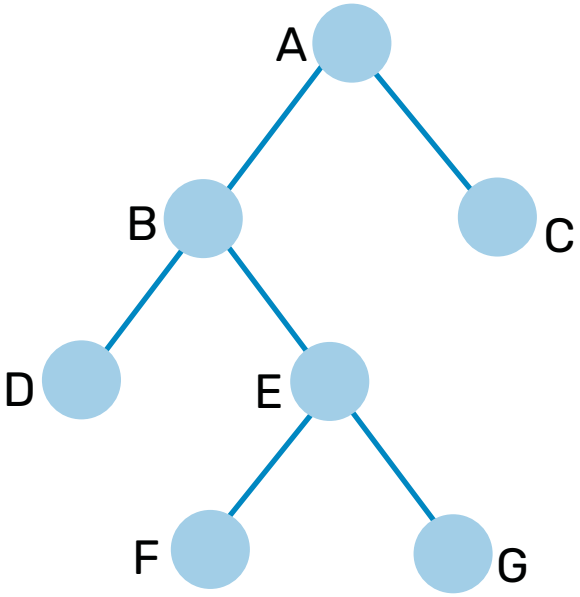


Figura 3 – Exemplo de árvore estritamente binária / Fonte: os autores.

Descrição da Imagem: exemplo de árvore binária com sete nós, na cor azul, interconectados por arestas (linhas). Fim da descrição.

A seguir, demonstramos estão algumas características significativas, que caracterizam uma árvore estritamente binária:

- **Cada nó interno tem exatamente dois filhos**: uma árvore estritamente binária tem dois nós internos. Isso significa que uma árvore com n nós internos terá exatamente 2n+1 nós totais, incluindo folhas.

- **Equilíbrio**: uma árvore estritamente binária pode ter um equilíbrio melhor que uma árvore binária normal. Entretanto, isso não é uma garantia. A construção da árvore determina o equilíbrio. Embora existam métodos para garantir o equilíbrio, como árvores balanceadas binárias, a condição binária por si só não garante um balanço ideal.
- **Aplicações**: árvores estritamente binárias funcionam bem para representar estruturas que precisam de uma distribuição bem definida de nós; exemplos disso incluem árvores de expressão, árvores de decisão e algumas variedades de árvores binárias de busca.

Como oferecem uma estrutura organizada e previsível, as árvores estritamente binárias, ou árvores binárias completas, têm muitas aplicações na ciência da computação e em áreas relacionadas. Vejamos, a seguir, algumas das principais funções dessas árvores.

Árvores de expressão

Árvores estritamente binárias podem ser usadas para representar expressões matemáticas ou lógicas em compiladores e linguagens de programação. Cada nó interno é um operador, enquanto os nós de folha são operandos ou valores.

Árvores de decisões

Árvores estritamente binárias são usadas para criar árvores de decisão em aprendizado de máquina e análise de dados. Cada nó interno representa uma condição ou pergunta, e as ramificações mostram quais decisões ou ações serão tomadas depois. Quando se trata de regressão e classificação, essa estrutura é útil.

Algoritmos binários de busca

As árvores de busca binárias (BST) organizam os dados de forma que a procura por um elemento seja rápida e eficiente. Estruturas mais complexas, como árvores AVL ou *Red-Black*, podem ser desenvolvidas a partir de BST estritamente binárias (Pereira, 2016).

Árvores de Huffman

Códigos de Huffman são criados por meio de árvores estritamente binárias na compressão de dados. Esse tipo de compressão sem perdas usa a frequência de ocorrência de um elemento para identificar como uma árvore se move.

Filas e pilares com prioridade

Filas de prioridade, em que os elementos são organizados com base em valores ou prioridades, são implementadas com árvores estritamente binárias, como árvores heap (*min-heap* ou *max-heap*). Em algoritmos de ordenação, como *heap sort*, e em situações em que você precisa acessar rapidamente o elemento de maior ou menor prioridade, isso é útil.

APROFUNDANDO

Aprendizagem de máquina, análise de dados, estatística e IA usam árvores de decisão. Em sua base, eles são usados para criar modelos de classificação e predição.

Uma árvore de decisão é uma estrutura hierárquica com nós. Cada nó interno representa uma decisão baseada em uma condição ou atributo, e cada ramo ou aresta representa o resultado dessa decisão. As saídas finais ou classes às quais os dados são atribuídos são representadas por nós de folha.

ÁRVORE BINÁRIA COMPLETA

O nó raiz de uma árvore binária é considerado como de Nível 0. A partir dele, cada nó possui um nível a mais do que o seu pai. Por exemplo, na árvore binária da Figura 4, o nó C está no Nível 1, F está no Nível 2 e o nó M, no Nível 3. A profundidade ou altura de uma árvore binária é dada pelo maior nível de qualquer folha na árvore. Isso equivale ao tamanho do percurso mais distante da raiz até uma folha qualquer. Dessa forma, a profundidade da árvore da Figura 3 é 3 (Pereira, 2016).

Quando uma árvore estritamente binária possui todas as folhas no último nível, ela é chamada de árvore binária completa. A Figura 4 demonstra uma árvore binária completa de profundidade 3.

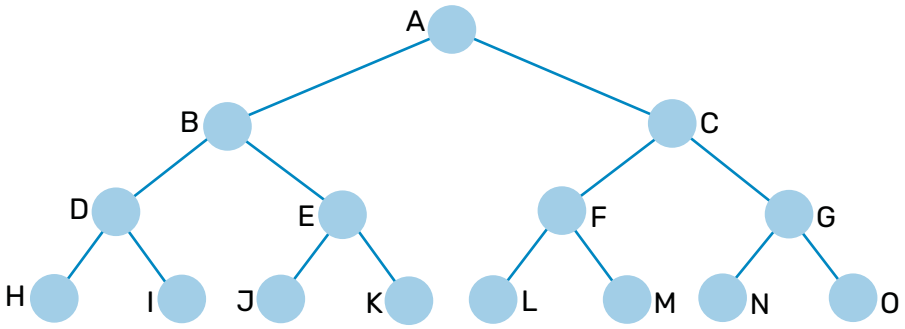


Figura 4: Exemplo de árvore binária completa / Fonte: o autor

Descrição da Imagem: Exemplo de árvore binária com dezesseis nós na cor azul interconectados por arestas (linhas). Fim da descrição.

A organização dos nós de uma árvore binária completa é sua característica distintiva:

- **Preenchimento por nível**: em uma árvore binária completa, todos os níveis anteriores ao último devem estar preenchidos completamente. Isso significa que cada nó interno deve ter dois filhos iguais.
- **Alinhamento à esquerda no nível mais alto**: o último nível requer que os nós sejam preenchidos o mais à esquerda possível, sem espaços entre eles. A árvore fica menor e menos alta por causa disso.

ÁRVORE BINÁRIA DIFERENTE

Outra forma de armazenar uma árvore binária em um vetor é reservar as posições de acordo com o nível e descendência de cada nó. O primeiro nó a ser armazenado é a raiz da árvore e ele ficará na primeira posição do vetor, lembrando que os vetores em C começam na posição 0 (Szwarcfiter; Markenzon, 2010).

Simulemos a inserção da árvore, representada pela Figura 4, num vetor de 16 posições. O primeiro nó a ser inserido será a raiz da árvore A, que ocupará a posição 0 do vetor.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A															

Como nosso objetivo é manter a árvore indexada dentro do vetor, reservaremos a primeira posição p logo após o nó para armazenar o filho esquerdo e a segunda posição p para o filho direito. Assim, para um nó armazenado numa posição p qualquer, seu filho esquerdo estará na posição p+1 e seu filho direito na posição p+2. Vamos inserir os nós B e C no vetor.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C													

O nó A está na posição p=0, assim, é esperado que seu filho esquerdo esteja na posição p+1=1 e o direito, na posição p+2=2, o que é verdade. Vamos inserir, agora, os nós D e E.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E											

Usando a fórmula que foi proposta, para encontrar os filhos de B na posição 1 do vetor, o filho esquerdo estará na posição p+1=2 e o direito, na posição p+2=3, o que não é verdade. Logo, a fórmula que propusemos não serve para resolver o problema de ordenação de árvore binária em vetor de dados.

Uma árvore binária cresce de forma geométrica posto que cada nó tem dois filhos, os quais, por sua vez, são duas subárvores que podem estar vazias ou não. Independentemente de o filho existir, seu espaço precisa ser reservado no vetor.

Como todo nó em uma árvore binária tem dois filhos, modifiquemos a fórmula para 2*p+1, para o filho esquerdo, e 2*p+2, para o filho direito, sendo p a posição do nó no vetor.

Aplicando as novas fórmulas para o nó A que está na posição 0, temos que 2*p+1=1 é a posição de B (filho esquerdo de A) e 2*p+2=2 é a posição de C (filho direito de A).

Para B, que está na posição 1, temos que 2*p+1=3, que é a posição de D (filho esquerdo de B) e 2*p+2=4 é a posição de D (filho direito de B).

O nó C é uma folha, podemos dizer que ele não tem filhos ou que seus filhos são árvores vazias. Como o nosso objetivo é manter o vetor ordenado, a posição dos filhos de C será reservada, aplicando a fórmula proposta. Assim, a posição 2*p+1=5 ficará disponível para o filho esquerdo de C e 2*p+2=6, para o seu filho direito.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	-	-									

O próximo será D, que também é uma folha. Assim, será reservado no veto as posições 2*p+1=7 e 2*p+2=8.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	-	-	-	-							

O último nó de nível 2 (E) possui um filho direito (F), que será armazenado na posição 2*p+2=10 do vetor criado para a nossa árvore binária.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	-	-	-	-	-	F					

No primeiro momento, podemos pensar que essa não é uma boa abordagem. Se a árvore não for binária completa, existirão vários espaços vazios no vetor, lembrando uma memória fragmentada. Por outro lado, ocuparemos exatamente a mesma quantidade de memória que a implementação anterior.

A principal diferença é que, nesse modelo, os nós da árvore estarão indexados, assim, é possível obter as informações de forma rápida e precisa, utilizando o índice em vez de percorrer toda a estrutura.

APROFUNDANDO

Quando falamos acerca de uma "árvore binária diferente", podemos nos referir a uma variedade de estruturas de árvore binária, que não seguem o padrão tradicional de árvores binárias, ou que implementam características únicas. Cada tipo de árvore oferece vantagens específicas dependendo do contexto e do problema a ser resolvido, proporcionando flexibilidade e eficiência para desenvolvedores e cientistas da computação.

EM FOCO

Acesse seu ambiente virtual de aprendizagem e confira a aula referente a este tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

NOVOS DESAFIOS

Para resolver o problema da organização de documentos na escola, visto anteriormente neste tema, sugere-se a implementação de uma árvore binária para organizar os arquivos digitais. Essa estrutura oferece eficiência na busca por documentos e facilita a inserção de novos itens, garantindo que o sistema permaneça equilibrado ao longo do tempo.

ESTRUTURA DA ÁRVORE BINÁRIA

A proposta é usar uma árvore binária de busca, na qual cada nó representa um documento ou conjunto de documentos. Para simplificar a busca, a árvore será organizada por data. O documento com a data mais antiga ficará em sua raiz e cada nó subsequente será inserido no lado esquerdo, se a data for mais antiga, ou no lado direito, se for mais recente.

PROCESSO DE BUSCA E INSERÇÃO

Para buscar um documento, o sistema percorre a árvore, começando pela raiz e seguindo para a esquerda ou direita, dependendo da data do documento desejado. Essa estrutura permite buscas rápidas, mesmo com um grande número de documentos. Para adicionar um novo documento, o sistema seguirá o mesmo processo, procurando a posição correta para inserir o novo nó de acordo com a data. Assim, a árvore permanece organizada e equilibrada.

BENEFÍCIOS DA SOLUÇÃO

Com essa solução baseada em árvores binárias, a escola poderá:

- Facilitar a busca por documentos, reduzindo o tempo necessário para encontrar informações.
- Simplificar a inserção de novos documentos, permitindo que a estrutura cresça conforme necessário.
- Manter a organização dos documentos de maneira intuitiva, facilitando a navegação para usuários menos experientes.
- Oferecer uma base sólida para expansão, caso a escola precise adicionar mais categorias ou tipos de documentos no futuro.

APLICAÇÕES FUTURAS

A solução proposta pode ser adaptada para outras necessidades da escola, como a organização de registros de alunos ou materiais de ensino. Além disso, a mesma abordagem pode ser aplicada em outras instituições que precisem de uma estrutura eficaz para organizar dados e informações.

De modo geral, o estudo de árvores binárias e suas variações é uma questão essencial em ciência da computação e programação.

A habilidade de compreender e trabalhar com estruturas de dados complexas é crucial no desenvolvimento de software e na ciência da computação. As árvores binárias estão entre as estruturas mais essenciais e são usadas em uma variedade de aplicações, incluindo a criação de algoritmos eficientes e organização de dados para busca e recuperação.

Quando os profissionais entendem como usar árvores binárias, eles são mais capazes de lidar com problemas difíceis e fazer uma contribuição significativa para suas equipes e organizações. Isso pode resultar em crescimento na carreira, cargos de liderança e participação mais ativa em projetos inovadores.

A capacidade de entender e aplicar ideias, como árvores binárias, é uma vantagem competitiva no mercado de trabalho em rápida mudança. As pessoas que trabalham com isso podem se adaptar a novas tecnologias, ajudar a desenvolver soluções inovadoras e participar ativamente do desenvolvimento de software e sistemas de alta qualidade.

Por isso, para quem deseja construir uma carreira sólida e duradoura no campo da tecnologia e da ciência da computação, é fundamental estudar árvores binárias e estruturas relacionadas.

VAMOS PRATICAR

1. Uma característica interessante das árvores binárias é que elas podem ser utilizadas para diversos propósitos em ciência da computação, como na implementação de algoritmos de busca e ordenação. Por exemplo, uma árvore binária de busca é um tipo específico de árvore binária em que todos os nós à esquerda de um nó possuem valores menores que o valor do nó em questão, e todos os nós à direita possuem valores maiores.

   Qual das seguintes afirmações é verdadeira acerca das árvores binárias?

   a) Em uma árvore binária, cada nó deve ter exatamente dois filhos.
   b) Uma árvore binária é uma estrutura de dados, que pode ser usada para representar hierarquias.
   c) Em uma árvore binária, todos os nós têm um número fixo de filhos.
   d) A árvore binária é um tipo de estrutura linear, que segue uma ordem sequencial.
   e) As árvores binárias não podem ser usadas para ordenar elementos ou representar relações de ordem.

2. Em ciência da computação, uma árvore estritamente binária é um tipo específico de árvore binária em que cada nó tem exatamente zero ou dois filhos. Em outras palavras, não existem nós com apenas um filho. Esse tipo de estrutura é usado em diversas aplicações, como árvores de busca binária, árvores de expressão, entre outras.

   Qual das seguintes características é verdadeira para árvores estritamente binárias?

   a) Em uma árvore estritamente binária, todos os nós têm pelo menos um filho.
   b) Uma árvore estritamente binária pode ter nós com apenas um filho.
   c) Em uma árvore estritamente binária, todos os nós têm exatamente dois filhos.
   d) Uma árvore estritamente binária pode ter nós folha que não têm filhos.
   e) Em uma árvore estritamente binária, a ordem dos filhos não é importante.

3. Uma árvore binária completa é uma estrutura de dados em que todos os níveis, exceto possivelmente o último, estão totalmente preenchidos, e todos os nós do último nível estão o mais à esquerda possível. Esse tipo de estrutura é amplamente usado em diversas aplicações, como em árvores binárias de busca, *heaps* e outras implementações.

   Com base nas informações apresentadas, avalie as asserções, a seguir, e a relação proposta entre elas:

   I - Uma árvore binária completa tem todos os níveis preenchidos com exceção do último, em que os nós são preenchidos da esquerda para a direita.

   PORQUE

   II - Numa árvore binária completa, a estrutura é otimizada para garantir que a altura seja minimizada enquanto todos os níveis acima do último estão completamente preenchidos.

   A respeito dessas asserções, assinale a opção correta:

   a) As asserções I e II são verdadeiras, e a II é uma justificativa correta da I.
   b) As asserções I e II são verdadeiras, mas a II não é uma justificativa correta da I.
   c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
   d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
   e) As asserções I e II são falsas.

REFERÊNCIAS

BACKES, A. R. **Algoritmos e estruturas de dados em linguagem C**. Rio de Janeiro: LTC, 2023.

OLIVEIRA, P. M. de, PEREIRA, R. de L. **Estrutura de dados II**. Maringá: Unicesumar, 2019.

PEREIRA, S. do L. **Estruturas de dados em C**: uma abordagem didática. São Paulo: Érica, 2016.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 2010.

CONFIRA SUAS RESPOSTAS

1. Alternativa B.

2. Alternativa D.

3. Alternativa A.

MEU ESPAÇO

TEMA DE APRENDIZAGEM 3

OPERAÇÕES SOBRE ÁRVORES BINÁRIAS

MINHAS METAS

- Compreender o caminhamento em árvores binárias.
- Entender a operação envolvida em árvores binárias – percurso pré-ordem.
- Entender a operação envolvida em árvores binárias – percurso em-ordem.
- Entender a operação envolvida em árvores binárias – percurso pós-ordem.
- Aprender o processo de busca em árvores binárias.
- Compreender as árvores AVL.
- Verificar, na prática, diferentes tipos de percursos em árvores.

INICIE SUA JORNADA

Uma empresa de comércio eletrônico opera um grande armazém que abriga milhares de produtos diferentes. Cada item tem um código numérico único que representa seu ID, além de outras informações, como nome, preço e quantidade em estoque. Para gerenciar o inventário de forma eficaz, a organização utiliza uma árvore binária de busca para manter e acessar rapidamente os dados dos produtos pelo seu ID.

Um problema crítico surge quando a empresa deseja:

- **Inserir um novo produto**: o código do novo produto deve ser adicionado à árvore binária no local correto para manter a ordenação e permitir pesquisas eficientes.
- **Buscar informações a respeito de um item**: dado um código de produto, encontrar rapidamente suas informações, como nome, preço e quantidade em estoque.
- **Remover um produto do inventário**: se um item é descontinuado ou não está mais em estoque, ele precisa ser removido da árvore.
- **Manter a árvore equilibrada**: com muitas operações de inserção e remoção, a estrutura pode se tornar desbalanceada, comprometendo a eficiência da busca.

Como programador, você foi encarregado de resolver essas questões.

A resolução desse problema é fundamental para a empresa manter um controle eficiente do inventário, permitindo que os clientes obtenham informações atualizadas acerca dos produtos em tempo real. Se a árvore binária for organizada corretamente, os processos de busca e atualização do inventário serão rápidos e eficazes, resultando em uma experiência positiva para os clientes e funcionários.

Se a árvore binária não for gerenciada adequadamente, por sua vez, poderá haver atrasos na busca de produtos, dificuldades na atualização do estoque e erros na remoção de produtos. Isso pode resultar em atrasos nas entregas, produtos fora de estoque e, por fim, em clientes insatisfeitos. Portanto, uma solução bem-implementada terá um impacto significativo no sucesso do negócio, garantindo operações mais fluidas e um serviço ao cliente de alta qualidade.

PLAY NO CONHECIMENTO

Neste episódio, discutiremos as árvores binárias, um assunto que é fundamental para qualquer desenvolvedor ou entusiasta de tecnologia.
Você usa as árvores binárias todos os dias, desde mecanismos de busca até controle de inventário em grandes empresas. Elas são essenciais para a busca, organização e gestão de dados. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

VAMOS RECORDAR?

Vamos nos lembrar das árvores binárias? Neste vídeo, discutiremos algumas das principais estratégias usadas e como elas impactam a eficácia da busca em grandes volumes de dados.

Acesse: **https://www.youtube.com/watch?v=VmKkAQtnjsM**

DESENVOLVA SEU POTENCIAL

Neste tema de aprendizagem, estudaremos como realizar certas operações em árvores binárias para solucionar problemas clássicos. Primeiramente, veremos como é possível gerar diferentes ordens de visitação nos nós de uma árvore por meio das técnicas de caminhamento em árvores binárias, a saber: pré-ordem, em-ordem e pós-ordem.

Em seguida, aprenderemos a criar uma árvore binária formatada especificamente para realizar a busca de elementos. Também, teremos uma noção de como manter tal estrutura balanceada por meio do algoritmo de Adelson-Velskii e Landis.

CAMINHAMENTO EM ÁRVORES BINÁRIAS

Em determinadas ocasiões, dependendo dos requisitos de uma aplicação, precisaremos percorrer todos os elementos de uma árvore para, por exemplo, exibir todo o seu conteúdo ao usuário. De acordo com a ordem de visitação dos nós, o usuário pode ter visões distintas de uma mesma árvore (Backes, 2023).

Imaginemos que, para percorrer uma árvore, tomamos o nó raiz como nó inicial e, a partir dele, começamos a visitar todos os nós adjacentes a ele para, só então, começar a investigar os outros nós da árvore. Suponhamos, também, que tomamos um nó folha como ponto de partida e caminhamos em direção à raiz, visitando apenas o ramo da árvore que leva o nó folha à raiz. São maneiras distintas de se visualizar a mesma estrutura.

Dito isso, vamos a alguns algoritmos clássicos que ditam as regras de visitação de nós individuais em uma árvore. Ao final da execução de cada um dos algoritmos a seguir, é esperado que sequências de visitação distintas sejam geradas.

Para tanto, considere que nossas árvores são compostas por registros dinâmicos em C, nos quais cada nó possui ao menos um ponteiro para a subárvore esquerda e outro, para a subárvore direita (Backes, 2023).

Percurso pré-ordem

O caminhamento pré-ordem, também conhecido por prefixado, marca primeiramente a raiz como visitada, somente depois, visitamos as subárvores esquerda e direita, respectivamente. O Programa 1, a seguir, apresenta um código-fonte em que a função preOrdem() implementa a lógica semântica necessária para fazer com que, a partir do parâmetro raiz, o programa realize o respectivo caminhamento em uma árvore binária.

```
1  void preOrdem(struct NO* raiz){
2        if(raiz){
3              printf("%d \t", raiz->dado); //visita o nó atual
4              preOrdem(raiz->esq);
5              preOrdem(raiz->dir);
6        }
7  }
```

Figura 1 – Programa 1 – função preOrdem() / Fonte: os autores.

Descrição da Imagem: a figura demonstra um algoritmo escrito na linguagem de programação C, ilustrando como acontece o percurso pré-ordem. Fim da descrição.

Considere o caminhamento pré-ordem na árvore da Figura 1, na qual a raiz é o nó F.

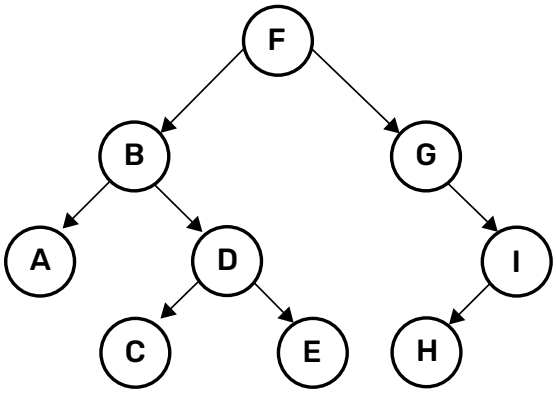


Figura 2 – Árvore binária / Fonte: os autores.

Descrição da Imagem: a figura representa uma árvore binária, composta de nove nós interligados por arestas, ou linhas. Fim da descrição.

A ordem de visitação produzida pela função preOrdem(), do Programa 1, levando em conta o nó inicial F, seria a seguinte: F, B, A, D, C, E, G, I, H.

APROFUNDANDO

O percurso em pré-ordem é útil para várias aplicações, incluindo:

- **Construção de árvores a partir de sua estrutura**: como o percurso em pré-ordem visita o nó antes de seus filhos, ele pode ser usado para descrever a estrutura geral de uma árvore, o que é útil para construção de árvores a partir de dados.
- **Expressão de árvores binárias como listas**: ele é útil para converter uma árvore em uma sequência linear, uma vez que preserva a hierarquia.
- **Outras operações recursivas**: o percurso em pré-ordem é usado como parte de outras operações mais complexas em árvores binárias, como clonagem e busca recursiva.

PERCURSO EM-ORDEM

No caminhamento em-ordem, também conhecido por interfixado, primeiramente, visitamos toda a subárvore esquerda e, só então, a raiz é marcada como visitada. Em seguida, o método em-ordem faz a visitação de toda a subárvore direita (Backes, 2023).

O Programa 2, a seguir, apresenta um código-fonte no qual a função emOrdem() implementa a lógica para fazer com que, a partir da raiz, o programa execute o caminhamento em-ordem em uma árvore binária.

```
1  void emOrdem(NO* raiz){
2          if(raiz){
3                  emOrdem(raiz->esq);
4                  printf("%d \t", raiz->dado); //visita o nó atual
5                  emOrdem(raiz->dir);
6          }
7  }
```

Figura 3 - Programa 2 - função em Ordem() / Fonte: os autores.

Descrição da Imagem: a figura demonstra um de algoritmo escrito na linguagem de programação C, ilustrando como acontece o percurso em-ordem. Fim da descrição.

Considere o caminhamento em-ordem na mesma árvore da Figura 1, vista anteriormente, em que a raiz é o nó F. A ordem de visitação produzida pela função emOrdem(), do Programa 2, levando em conta o nó inicial F, seria a seguinte: A, B, C, D, E, F, G, H, I.

Percurso pós-ordem

O último caminho que veremos é o método pós-ordem, também conhecido por pós-fixado. Nesse caso, primeiramente, visitamos toda a subárvore esquerda, depois, toda a subárvore direita. Só após ter visitado as duas subárvores, é que marcamos o nó corrente como visitado. O Programa 3, a seguir, apresenta a função posOrdem(), que faz com que, a partir da raiz, o programa execute o caminhamento pós-ordem em uma árvore binária (Oliveira; Pereira, 2019).

```
1  void posOrdem(NO* raiz){
2          if(raiz){
3                  posOrdem(raiz->esq);
4                  posOrdem(raiz->dir);
5                  printf("%d \t", raiz->dado); //visita o nó atual
6          }
7  }
```

Figura 4 – Programa 3 – função posOrdem() / Fonte: os autores.

Descrição da Imagem: a figura demonstra um algoritmo escrito na linguagem de programação C, ilustrando como acontece o percurso pós-ordem. Fim da descrição.

Considerando o caminhamento pós-ordem na árvore já conhecida, da Figura 1, a ordem de visitação produzida pela função posOrdem(), do Programa 3, levando em conta o nó inicial F, seria a seguinte: A, C, E, D, B, H, I, G, F.

APROFUNDANDO

O percurso pós-ordem é especialmente útil em diversas situações, como:

- **Deleção em árvores binárias**: quando se remove um nó, geralmente, é necessário percorrer primeiro as subárvores para garantir uma remoção segura.
- **Avaliação de expressões matemáticas**: o percurso pós-ordem é usado para calcular valores de árvores de expressão, garantindo que todos os operandos sejam avaliados antes do operador.
- **Liberação de recursos**: em operações que exigem a limpeza ou desalocação de recursos em árvores binárias, o percurso pós-ordem garante que todos os descendentes sejam liberados antes do pai, evitando problemas, como vazamentos de memória.

BUSCA EM ÁRVORES BINÁRIAS

Árvores binárias são muito utilizadas para organizar informações na memória devido ao seu grande potencial de busca em um tempo relativamente curto. Para isso, precisamos criar uma árvore binária de busca (Pereira, 2016).

Você já sabe como criar um vetor a partir de uma árvore dada. Vamos fazer exatamente o contrário. A partir de um vetor, construiremos uma árvore binária com regras específicas para que ela se torne uma estrutura de busca.

Uma árvore binária ou é vazia ou tem pelo menos um nó raiz. Todo nó tem um pai (menos a raiz) e no máximo dois filhos, um esquerdo e um direito. Tanto o filho esquerdo como o direito podem ser subárvores vazias.

Vamos, agora, adicionar uma nova regra para a construção da árvore. O nó raiz será o valor que estiver na posição do meio de um vetor (ordenado ou não). Ao adicionar um novo nó na árvore, verificamos se ele é menor do que a raiz. Caso seja verdade, ele será adicionado na subárvore esquerda; caso contrário, na subárvore direita. Faremos uma simulação, usando como base um vetor ordenado representado na Figura 2. A primeira linha da figura é o índice do vetor, que vai de 0 a 9. A segunda linha possui os valores de cada posição do vetor (Szwarcfiter; Markenzon, 2010).

0	1	2	3	4	5	6	7	8	
3	1	8	7	20	21	31	40	30	

Figura 5 - Vetor para construção de uma árvore binária
Fonte: os autores.

Descrição da Imagem: a tabela contém 10 elementos, começando pelo elemento 0 indo até o elemento 9. Cada um deles possui um número aleatório. Fim da descrição.

Como os valores de índices de um vetor são inteiros, meio será 4. O valor do vetor na posição 4 (vec[4]) será a raiz da árvore.

Figura 6 - Raiz de uma árvore / Fonte: os autores.

Descrição da Imagem: círculo em azul com o número 20 ao centro, representando o início de uma árvore. Fim da descrição.

Agora que a nossa árvore possui uma raiz, adicionaremos o primeiro elemento do vetor na árvore. Como vec[0] é menor do que a raiz, seu valor será inserido na subárvore esquerda.

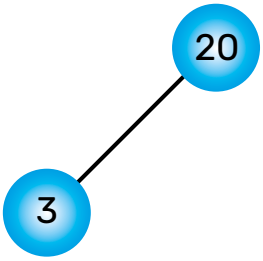


Figura 7 - Árvores e suas ramificações / Fonte: os autores.

Descrição da Imagem: círculo em azul com o número 20 ao centro interligado com o número 3, representando o início de uma árvore. Fim da descrição.

O próximo valor vec[1] também é menor do que a raiz, mas a raiz já possui uma subárvore esquerda. Fazemos, então, uma nova comparação com a raiz da subárvore esquerda. Como vec[1] é menor do que a raiz dessa subárvore, ele será adicionado como seu filho esquerdo.

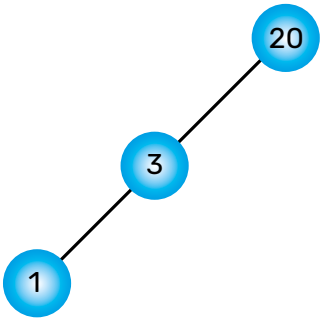


Figura 8 – Árvores e suas ramificações / Fonte: os autores.

Descrição da Imagem: círculo em azul com o número 20 ao centro interligado com o número 3 e com o número 1, representando o início de uma árvore. Fim da descrição.

Dando sequência, verificamos o valor contido em vec[2]. Seu valor é menor do que 20, então, checamos com o filho esquerdo da raiz. O valor de vec[2] é maior do que 3, assim, ele entrará na árvore como seu filho direito.

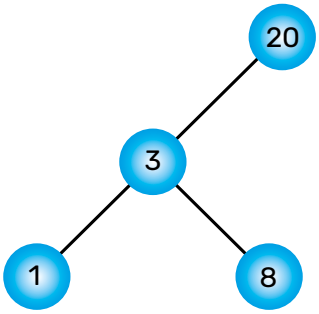


Figura 9 – Árvores e suas ramificações e estágio avançado
Fonte: os autores.

Descrição da Imagem: círculo em azul com o número 20 ao centro interligado com o número 3 e com o número 1. O número 3, por sua vez, está interligado com o número 8. Essa figura representa uma árvore em estágio avançado de conclusão. Fim da descrição.

Agora, é a vez de vec[3], que tem valor 7. Percorremos a árvore em formação, respeitando as regras da árvore binária de busca e um novo nó será adicionado como filho esquerdo de 8.

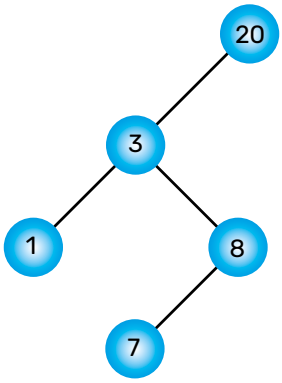


Figura 10 – Árvores e suas ramificações e estágio avançado
Fonte: os autores.

Descrição da Imagem: círculo em azul com o número 20 ao centro interligado com o número 3 e com o número 1. O número 3, por sua vez, está interligado com o número 8. O número 8 está conectado, agora, com o número 7. Essa figura representa uma árvore em estágio avançado de conclusão. Fim da descrição.

Chegamos, agora, à metade do vetor da Figura 2. Precisamos de mais cinco interações para finalizar a construção da árvore. A Figura 3 apresenta a árvore final após a inserção dos nós de valores 21, 31, 40, 30 e 0.

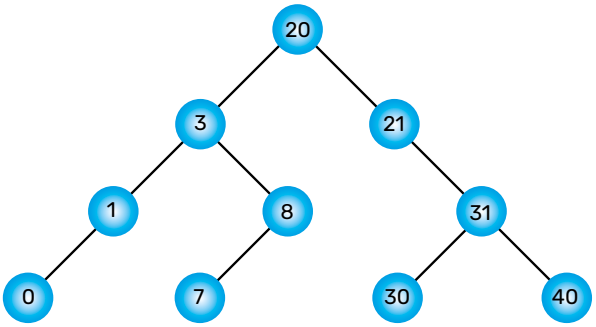


Figura 11 – Árvores e suas ramificações e estágio avançado
Fonte: os autores.

Descrição da Imagem: árvore completa, com 10 nós interconectados por arestas, ou linhas. Fim da descrição.

Agora vem o que nos interessa, que é a realização da busca. Dado um argumento qualquer, se ele for menor do que a raiz, ou ele não existe, ou ele se encontra na sua subárvore esquerda. Se o valor for maior do que a raiz, ou ele não existe, ou está na sua subárvore direita. A partir da raiz, descemos pela árvore binária de busca até que o valor seja encontrado ou que achemos uma folha ou uma subárvore vazia.

A Figura 13 apresenta uma busca na árvore pelo valor 7. A raiz tem valor 20, assim, a busca prossegue na sua subárvore esquerda. Como 7 > 3, a busca contínua pela subárvore direita do nó 3. Como 7 < 8, a busca desce pela subárvore esquerda do nó 8 até que o valor 7 seja encontrado.

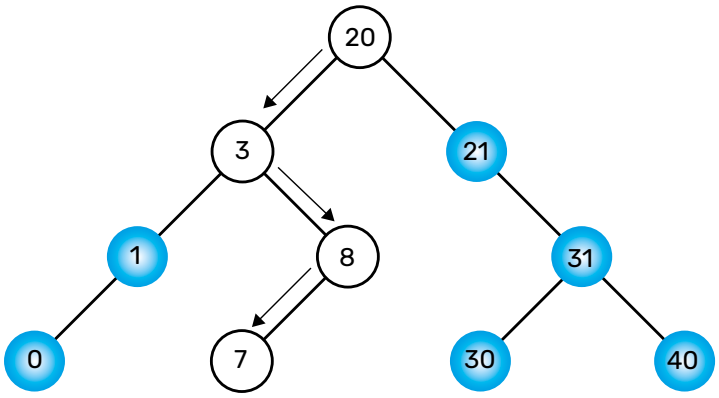


Figura 12 – Buscas em árvores / Fonte: os autores.

Descrição da Imagem: a figura mostra que a busca começa pelo nó de número 20, percorrendo uma das ramificações até chegar no número 3. Fim da descrição.

Essa técnica se assemelha muito à busca binária. A principal diferença é que a árvore binária de busca pode ser implementada dinamicamente na memória, pois não precisamos descobrir em qual local fica a metade da tabela. Em vez de dividir o vetor pela metade, ignoramos uma das subárvores para continuar a pesquisa. Esse método é extremamente rápido e eficiente em arquivos com grandes quantidades de registros.

O Programa 4 demonstra um algoritmo em linguagem C, que encontra um valor por meio de uma árvore binária de busca em um vetor de dados. A função buscaArvoreBinaria() recebe três parâmetros: o vetor vec a ser pesquisado, o argumento arg a ser encontrado e o tamanho tam do vetor.

O laço principal tem duas regras de parada. O laço continua até o valor ser encontrado, que se dá no momento em que a variável achou for diferente de -1 ou quando chegar ao final do vetor.

Para ficar mais simples a implementação, utilizamos um vetor que guarda os nós de forma ordenada de acordo com o nível de cada nó. Assim, a busca começa na raiz e segue para 2*p+1 na direção da árvore esquerda ou 2*p+2 na árvore direita, em que p é a posição do nó no vetor.

```
//Função de árvore binária de busca
int buscaArvoreBinaria(int vec[], int arg, int tam){
  int no, achou, fim;
  fim = 0;
  no = 0;
  achou = -1;
  while((achou == -1) && (fim <= tam)){
    if (arg == vec[no]){
      achou = no;
    }
    if (arg < vec[no]){
      no = (2 * no) + 1;
    }
    else {
      no = (2 * no) + 2;
    }
    fim++;
  }
  return(achou);
}
```

Figura 13 - Programa 4 - função de árvore binária de busca
Fonte: os autores.

Descrição da Imagem: o algoritmo, na linguagem de programação C, representa a busca em árvore binária. Fim da descrição.

EM FOCO

Acesse seu ambiente virtual de aprendizagem e confira a aula referente a este tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

NOVOS DESAFIOS

As árvores binárias são frequentemente usadas em vários contextos tecnológicos, incluindo bancos de dados, estruturas de dados para mecanismos de busca, compiladores, gerenciamento de arquivos, sistemas de recomendação e muito mais, devido à resolução e situação desses problemas. Um bom entendimento dessas estruturas permite a criação de soluções eficazes para problemas difíceis.

Garantir o desempenho das árvores binárias é uma das principais preocupações ao trabalhar com elas. As estruturas desbalanceadas podem causar operações de busca e inserção muito complexas, o que leva a um sistema lento e ineficiente. Para evitar esses problemas, os profissionais de tecnologia devem saber como manter uma árvore equilibrada. A eficiência depende de métodos, como rotações e árvores balanceadas, como as Árvores AVL.

Para resolver o problema do controle de inventário em um armazém, você deve implementar operações-chave em uma árvore binária de busca. Essas operações facilitarão a inserção, busca e remoção de produtos. As rotas para cada operação, sem entrar em detalhes de código, estão aqui:

INTRODUZIR UM NOVO PRODUTO

Para inserir um novo produto na árvore binária, você deve verificar o ID do novo nó com o dos nós existentes. O produto é colocado na subárvore à esquerda, se for menor, e na subárvore à direita, se for maior. Continue com essa comparação até encontrar a posição de inserção adequada.

ENCONTRAR UM PRODUTO
Para buscar um produto por seu ID, use a mesma lógica de comparação: se o ID buscado for menor que o nó atual, ele se move para a esquerda, e se for maior, ele se move para a direita. Continue até encontrar o produto ou até chegar ao fim da árvore.
REMOVER UM PRODUTO
Primeiramente, encontre o nó correspondente na árvore para remover um produto pelo ID. Dependendo de quantos filhos ele tenha, a remoção pode ser fácil ou requer reorganização.
ACOMPANHAR O NÓ FILHO
Reorganize a árvore para manter a estrutura de busca se tiver dois filhos. O menor nó da subárvore à direita deve ser substituído pelo sucessor.

Com a resolução bem-sucedida dessa situação-problema, o controle de inventário no armazém é mais rápido e eficiente. Isso facilita operações, como buscar produtos, adicionar novos itens e remover os que não estão disponíveis. Melhorando a experiência do cliente e a eficiência do armazém, as árvores binárias equilibradas garantem que o desempenho permaneça alto mesmo com muitas operações. O desbalanceamento da árvore pode levar a atrasos e problemas no gerenciamento do inventário, prejudicando a eficiência geral do negócio.

Sistemas escaláveis e eficazes capazes de resolver uma variedade de problemas computacionais podem ser criados com base em uma boa compreensão das operações das árvores binárias. Profissionais de tecnologia, que buscam construir soluções sólidas e seguras, precisam ter a capacidade de manter o desempenho ao longo do tempo, ao mesmo tempo em que estão atentos a questões éticas e de privacidade. A consideração desses elementos guiará o desenvolvimento de software em direção a práticas melhores e mais eficientes que beneficiarão tanto os usuários quanto as organizações.

VAMOS PRATICAR

1. Um percurso de pré-ordem em árvores binárias é uma técnica de travessia na qual o nó raiz é visitado primeiro, seguido pelos filhos à esquerda e depois pelos filhos à direita. Esse tipo de percurso é útil para várias aplicações, como reconstrução de árvores, análise de estrutura hierárquica, entre outras.

   Considere uma árvore binária em que cada nó contém um valor único. Qual seria a sequência correta para uma travessia de pré-ordem?

   a) Visitar todos os nós folhas primeiro e depois, os nós internos.
   b) Visitar o nó raiz e depois, todos os filhos à esquerda e todos os filhos à direita.
   c) Visitar o nó raiz, em seguida, os filhos à esquerda, os filhos dos nós à esquerda e os filhos à direita e, por fim, os filhos dos nós à direita.
   d) Visitar o nó raiz, em seguida, o filho esquerdo, o filho direito, o filho do filho esquerdo e, assim por diante, sempre da esquerda para a direita.
   e) Visitar o nó raiz, em seguida, o filho esquerdo, o filho direito, os subfilhos da esquerda para a direita até completar a travessia.

2. Nas ciências da computação, a travessia de pós-ordem em árvores binárias é uma técnica de percurso em que os filhos de um nó são visitados antes do próprio nó. A ordem do percurso é a seguinte: primeiro são visitados todos os filhos à esquerda, depois os filhos à direita, e, por fim, o nó raiz. Essa técnica é frequentemente usada em algoritmos de avaliação de expressões aritméticas e para liberar recursos de maneira segura em estruturas de dados.

   Considere uma árvore binária em que cada nó contém um valor único. Qual seria a sequência correta para uma travessia de pós-ordem?

   a) Visitar o nó raiz, em seguida, todos os filhos à esquerda e todos os filhos à direita.
   b) Visitar todos os nós folhas primeiro, em seguida, os nós internos e, por fim, o nó raiz.
   c) Visitar o nó raiz, em seguida, o filho esquerdo e o filho direito.
   d) Visitar todos os filhos à esquerda, em seguida, todos os filhos à direita e, por fim, o nó raiz.
   e) Visitar todos os filhos à direita, depois todos os filhos à esquerda antes de visitar o nó raiz.

3. O percurso de pós-ordem em árvores binárias é uma técnica de travessia em que os nós são visitados após seus filhos. Primeiramente, são visitados todos os filhos à esquerda, depois todos os filhos à direita, e, finalmente, o nó raiz. Esse método é usado em várias aplicações, como avaliação de expressões aritméticas, geração de código para compiladores e liberação segura de memória em estruturas de dados.

   Com base nas informações apresentadas, avalie as asserções, a seguir, e a relação proposta entre elas:

   I - No percurso de pós-ordem em árvores binárias, o nó raiz é visitado depois dos filhos à esquerda e à direita.

   PORQUE

   II - No percurso de pós-ordem, a sequência de travessia é escolhida para assegurar que todas as subárvores sejam completamente processadas antes de visitar o nó raiz, garantindo uma ordem lógica para operações como exclusão ou avaliação.

   A respeito dessas asserções, assinale a opção correta:

   a) As asserções I e II são verdadeiras, e a II é uma justificativa correta da I.
   b) As asserções I e II são verdadeiras, mas a II não é uma justificativa correta da I.
   c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
   d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
   e) As asserções I e II são falsas.

REFERÊNCIAS

BACKES, A. R. **Algoritmos e estruturas de dados em linguagem C**. Rio de Janeiro: LTC, 2023.

OLIVEIRA, P. M. de, PEREIRA, R. de L. **Estrutura de dados II**. Maringá: Unicesumar, 2019.

PEREIRA, S. do L. **Estruturas de dados em C**: uma abordagem didática. São Paulo: Érica, 2016.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 2010.

CONFIRA SUAS RESPOSTAS

1. Alternativa E.

2. Alternativa D.

3. Alternativa A.

MEU ESPAÇO

Unicesumar

unidade 3

TEMA DE APRENDIZAGEM 4

EXCLUSÃO EM ÁRVORES BINÁRIAS

MINHAS METAS

- Compreender conceitos iniciais de exclusão em árvores binárias.
- Entender a operação de exclusão em árvores binárias.
- Compreender como funciona a estruturação de exclusão.
- Verificar como acontece a complexidade de tempo em exclusão.
- Entender os principais algoritmos de exclusão.
- Verificar, na prática, o processo de balanceamento de árvores.
- Observar, na prática, o processo de exclusão.

INICIE SUA JORNADA

Imagine que você está desenvolvendo um aplicativo de gerenciamento de estoque para uma empresa de varejo. Você precisa implementar um sistema que permita a exclusão de produtos do estoque, quando eles não estão mais disponíveis para venda. Entretanto, você percebe que a exclusão desses itens pode afetar a integridade dos dados e a eficiência do seu aplicativo. Como garantir que a exclusão das mercadorias do estoque seja realizada de forma correta e eficiente?

A resolução adequada desse problema é importante para a integridade dos dados e o bom funcionamento do seu aplicativo de gerenciamento de estoque. Uma exclusão incorreta pode resultar em inconsistências nos registros, levando a erros na contagem de itens disponíveis e comprometendo a precisão das análises de vendas e reposição de estoque. Além disso, uma exclusão ineficiente pode impactar negativamente o desempenho do aplicativo, tornando as operações de exclusão demoradas e afetando a experiência do usuário.

PLAY NO CONHECIMENTO

Você está preparado para compreender o processo de exclusão em árvores binárias? Neste episódio, desvendamos os desafios e exploramos estratégias essenciais para realizar essa atividade. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

VAMOS RECORDAR?

Vamos recordar as árvores binárias?

No vídeo, a seguir, há um resumo a respeito desse tema. Esse material é fundamental não apenas para compreender os desafios, mas também para desenvolver estratégias eficazes que permitam a exclusão de dados em uma determinada árvore.

Acesse: **https://youtube.com/watch?v=VmKkAQtnjsM**.

DESENVOLVA SEU POTENCIAL

EXCLUSÃO EM ÁRVORES BINÁRIAS

As árvores binárias são estruturas de dados importantes que consistem em nós conectados por arestas e podem ter até dois filhos por nó. O processo de exclusão em árvores binárias é fundamental para manipular e alterar essas estruturas dinamicamente, removendo nós específicos.

A teoria das árvores binárias existe desde o início da ciência da computação, na década de 1960. Embora a pesquisa inicial tenha se concentrado, principalmente, em algoritmos de inserção e busca, ela rapidamente se expandiu para incluir expressões de exclusão. Desde então, uma grande quantidade de algoritmos foi desenvolvida e refinada para realizar uma exclusão eficaz em árvores binárias (Backes, 2023).

Considerando árvores binárias, o processo de exclusão é uma operação frequentemente necessária para manipular e alterar a estrutura da árvore de acordo com as necessidades específicas de uma aplicação. O processo de exclusão em árvores binárias consiste na remoção de um nó específico de uma árvore. Quando o nó a ser removido tem filhos, esse processo pode ser mais complicado (Backes, 2023).

APROFUNDANDO

Árvores binárias têm uma ampla gama de aplicações em ciência da computação e engenharia de software devido à sua estrutura flexível e eficiente. Aqui, estão algumas das principais aplicações:

- **Bancos de dados**: em sistemas de gerenciamento de banco de dados, árvores binárias são usadas para implementar índices que facilitam a rápida recuperação de dados. As árvores binárias de busca são especialmente úteis aqui, pois permitem pesquisas eficientes e ordenadas.
- **Pesquisa e ordenação**: árvores binárias de busca são amplamente utilizadas em algoritmos de pesquisa e ordenação, como a pesquisa binária e a árvore de busca binária balanceada (AVL). Essas estruturas de dados oferecem tempos de execução eficientes para operações de inserção, exclusão e pesquisa, tornando-as ideais para aplicativos que exigem acesso rápido aos dados.

Vejamos como acontece esse processo:

1. **Identifique quem é excluído**: encontrar o nó que você deseja remover da árvore é o primeiro passo. As regras de busca, na árvore binária, podem ser usadas para esse propósito até encontrar o nó pretendido.
2. **Caso 1**: nó folha: a exclusão é bastante fácil se o nó que você deseja excluir for uma folha e não tiver filhos. Basta mudar o ponteiro do pai para *null* (ou seja, definindo o ponteiro do pai para o nó excluído como *null*).
3. **Caso 2**: nó com um filho: se o nó que você deseja excluir tiver apenas um filho, a exclusão também é direta. Você pode substituir o nó a ser excluído pelo seu único filho. Esse processo envolve ajustar o ponteiro do pai do nó a ser excluído para apontar para o filho do nó a ser excluído.
4. **Caso 3: nó com dois filhos**: se o nó que você deseja excluir tiver dois filhos, o processo é mais complexo. Nesse caso, você precisa encontrar o sucessor do nó a ser excluído. O sucessor é o nó com o próximo valor mais alto na árvore. Isso geralmente é o nó mais à esquerda da subárvore direita do nó a ser excluído. Após encontrar o sucessor, você o substitui pelo nó a ser excluído e, então, exclui o sucessor original.
5. **Remover o nó sucessor**: depois de substituir o nó a ser excluído pelo seu sucessor, você precisa remover o sucessor original de sua posição original na árvore. Dependendo dos casos anteriores, você pode aplicar a lógica de exclusão apropriada para o nó sucessor.
6. **Reorganizando os ponteiros**: para garantir que a estrutura da árvore seja mantida corretamente, os ponteiros devem ser reorganizados após a exclusão do nó e, se necessário, a remoção do sucessor. Isso pode incluir alterações nos ponteiros do pai, filho e, se aplicável, do nó sucessor.
7. **Verificação e atualização da propriedade da árvore**: após a exclusão, é essencial verificar se a propriedade da árvore binária (por exemplo, uma árvore de busca binária deve permanecer ordenada) ainda é mantida. Pode ser necessário que você realize operações adicionais para reequilibrar a árvore ou garantir que outras propriedades sejam mantidas (Oliveira; Pereira, 2019).

Em árvores binárias, a operação de exclusão é essencial para preservar a estrutura da árvore enquanto os dados são inseridos, atualizados ou removidos. Para desenvolver algoritmos que envolvem árvores binárias de forma eficiente, é necessário dominar esse processo.

APROFUNDANDO

Em compiladores e interpretadores, as árvores binárias são usadas para representar a estrutura sintática do código-fonte durante o processo de análise. Essas árvores, conhecidas como árvores de análise sintática ou árvores de sintaxe abstrata, são usadas em compiladores e interpretadores para gerar código intermediário ou executar diretamente o código-fonte.

Já em redes de computadores e em protocolos de comunicação de rede, árvores binárias são usadas para implementar algoritmos de roteamento eficientes. Por exemplo, em árvores de roteamento, cada nó representa um roteador na rede e as arestas representam as conexões entre os roteadores.

Árvores binárias são frequentemente utilizadas como estruturas de dados auxiliares em uma variedade de algoritmos e aplicações. Por exemplo, árvores binárias podem ser usadas para implementar filas de prioridade em algoritmos de busca de caminho mais curto, como o algoritmo de Dijkstra (Pereira, 2016).

Que tal vermos esse processo de exclusão a partir de um exemplo prático? Consideremos a seguinte árvore:

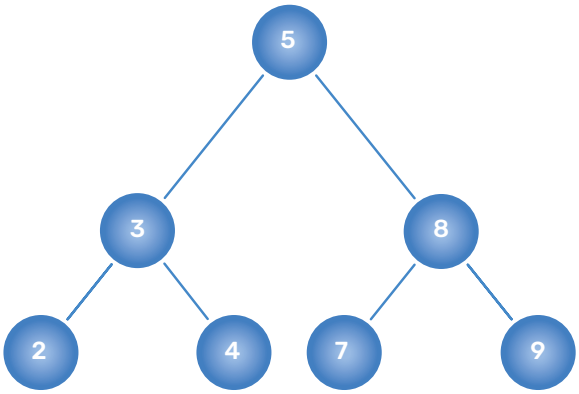


Figura 1 – Processo de exclusão em árvores binárias
Fonte: os autores.

Descrição da Imagem: a figura demonstra uma árvore interligada, apresentando sete nós interligados por arestas, ou linhas. Fim da descrição..

- **Passo 1**: identificar o nó a ser excluído: o nó a ser excluído é o que possui valor 3.
- **Passo 2**: determinar o caso: o nó 3 possui dois filhos; trata-se, portanto, do Caso 3.
- **Passo 3**: encontrar o sucessor: para encontrar o sucessor do nó 3, devemos procurar o menor valor na subárvore direita. Nesse caso, o sucessor de 3 é o nó 4.

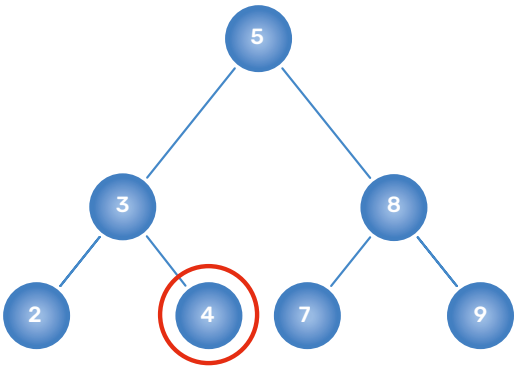


Figura 2 – Processo de exclusão em árvores binárias
Fonte: os autores.

Descrição da Imagem: a figura mostra uma árvore interligada, apresentando sete nós interligados por arestas, ou linhas. Fim da descrição.

- **Passo 4**: substituir o nó 3 pelo seu sucessor (4): agora, substituímos o valor do nó 3 pelo valor do sucessor, que é 4. Além disso, devemos manter a estrutura da árvore, garantindo que os filhos do nó 3 permaneçam conectados corretamente.

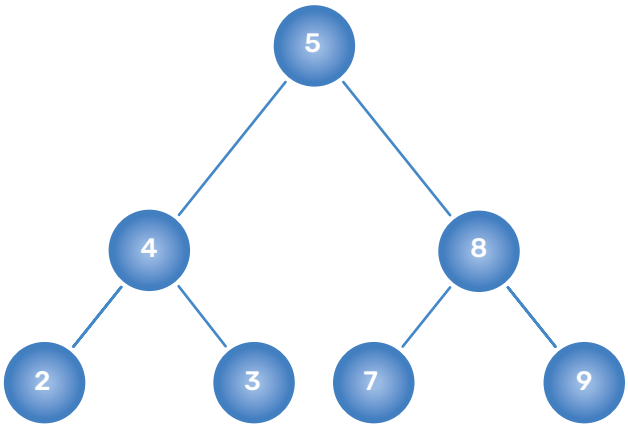


Figura 3 – Processo de exclusão em árvores binárias
Fonte: os autores.

Descrição da Imagem: a figura demonstra uma árvore interligada, apresentando sete nós interligados por arestas, ou linhas. Fim da descrição.

- **Passo 5**: remover o sucessor original (4): como o nó sucessor 4, agora, foi movido para a posição do nó 3, o nó sucessor original não está mais presente na árvore. Não precisamos fazer mais nada em relação a ele.
- **Passo 6**: reorganizar ponteiros (se necessário): nesse caso, não há necessidade de reorganizar ponteiros, pois apenas alteramos os valores dos nós, mantendo a estrutura da árvore. A árvore resultante não contém mais o nó com valor 3.

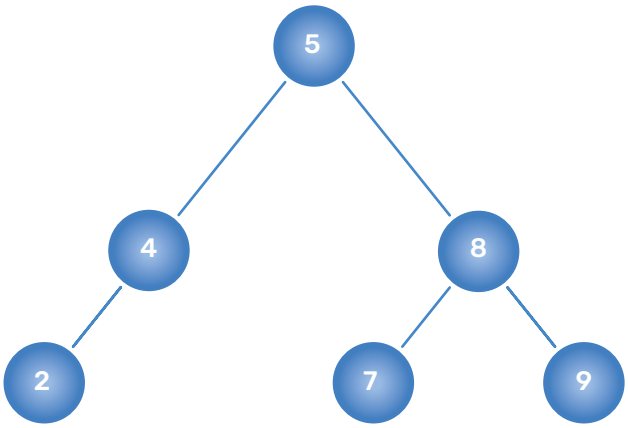


Figura 4 – Processo de exclusão em árvores binárias
Fonte: os autores.

Descrição da Imagem: a figura demonstra uma árvore interligada, apresentando sete nós interligados por arestas, ou linhas. Fim da descrição.

Esse exemplo ilustra um cenário simples de exclusão em uma árvore binária. Em situações mais complexas, como quando o nó a ser excluído possui apenas um filho ou é uma folha, os passos seriam ajustados de acordo.

PRINCIPAIS ALGORITMOS PARA EXCLUSÃO

Existem vários algoritmos para exclusão em árvores binárias, cada um com suas características e complexidades. A seguir, elencamos alguns dos principais algoritmos para exclusão em árvores binárias (Szwarcfiter; Markenzon, 2010).

Exclusão simples (sem reequilíbrio)

Trata-se do algoritmo mais simples e aplicável quando a árvore não precisa ser equilibrada após a exclusão. Consiste na identificação do nó a ser excluído e ajuste dos ponteiros do pai para apontar para os filhos do nó a ser excluído. É adequado para árvores binárias simples, mas pode resultar em árvores não balanceadas em algumas situações.

Exclusão com reequilíbrio (por exemplo, AVL ou árvores vermelho-negras)

Em árvores balanceadas, como árvores AVL ou árvores vermelho-negras, a exclusão requer reequilíbrio para manter a altura balanceada da estrutura. Após a exclusão, é necessário verificar e, se necessário, ajustar as alturas e as cores dos nós para garantir que as propriedades da árvore sejam mantidas. Esse processo pode envolver rotações e recolocação de nós para garantir que a árvore permaneça balanceada.

Exclusão com sucessor (ou predecessor)

O algoritmo envolve encontrar o sucessor (ou predecessor) do nó a ser excluído e substituí-lo pelo nó a ser excluído. Em seguida, o nó sucessor (ou predecessor) original é removido da árvore. Esse método é comumente usado, quando o nó a ser excluído possui dois filhos.

Exclusão recursiva

Trata-se do algoritmo que envolve a exclusão de nós de forma recursiva, percorrendo a árvore até encontrar o nó a ser excluído. Após a exclusão do nó desejado, a função recursiva é chamada para os filhos do nó retirado, se houver. Esse método é eficiente e elegante, mas requer cuidado para garantir que todos os ponteiros sejam atualizados corretamente.

Exclusão iterativa

O algoritmo utiliza um loop iterativo para percorrer a árvore e encontrar o nó a ser excluído. Uma vez que o nó é encontrado, os ponteiros são ajustados conforme necessário para excluir o nó da árvore. Esse método é mais direto e pode ser mais eficiente em algumas situações, especialmente, para árvores binárias simples.

Esses são apenas alguns dos principais algoritmos para exclusão em árvores binárias. A escolha do algoritmo depende das características da árvore (como balanceamento) e dos requisitos de desempenho da aplicação.

EM FOCO

Acesse seu ambiente virtual de aprendizagem e confira a aula referente a este tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

NOVOS DESAFIOS

Para resolver a questão proposta no início deste tema, você pode experimentar diferentes abordagens de exclusão em árvores binárias. Pode, por exemplo, implementar um algoritmo de eliminação que encontre o sucessor de um nó a ser excluído e o substitua, garantindo que a estrutura da árvore seja mantida corretamente. Além disso, pode explorar técnicas de remoção iterativa ou recursiva para lidar com diferentes casos de exclusão, como nós com um ou nenhum filho. Ao experimentar essas abordagens na prática, você poderá identificar a melhor solução para o seu aplicativo de gerenciamento de estoque.

Ao lidar com a exclusão em árvores binárias em seu projeto de desenvolvimento de software, é importante refletir acerca da importância de se entender os conceitos fundamentais por trás dessa operação. A exclusão em árvores binárias não se limita apenas ao contexto de desenvolvimento de aplicativos de gerenciamento de estoque, mas é uma habilidade essencial que pode ser aplicada em uma variedade de cenários na ciência da computação. Ao dominar essa técnica, você estará mais bem-preparado para enfrentar desafios em projetos futuros e contribuir para o desenvolvimento de soluções robustas e eficientes.

VAMOS PRATICAR

1. As árvores binárias são estruturas de dados importantes nas ciências da computação, compostas por nós conectados por arestas, em que cada nó pode ter até dois filhos. Uma operação fundamental em árvores binárias é a exclusão de nós, que permite alterar dinamicamente a estrutura da árvore, removendo elementos específicos.

   Qual é a importância do processo de exclusão em árvores binárias?

   a) A exclusão é necessária para adicionar novos nós à árvore binária.
   b) A exclusão permite a remoção de nós específicos da árvore, alterando sua estrutura dinamicamente.
   c) A exclusão garante que a árvore binária tenha sempre no máximo dois filhos por nó.
   d) A exclusão simplifica a operação de pesquisa em árvores binárias.
   e) A exclusão em árvores binárias não é uma operação significativa.

2. Na manipulação de árvores binárias, a exclusão de nós desempenha um papel fundamental na manutenção e na alteração da estrutura da árvore. Quando o nó a ser excluído é uma folha, ou seja, não possui filhos, o processo de exclusão é relativamente simples.

   Considerando o contexto da exclusão de nós em árvores binárias, qual é o procedimento para excluir um nó folha?

   a) Remover todos os nós abaixo do nó folha.
   b) Redefinir o valor do nó folha como nulo.
   c) Alterar o ponteiro do pai do nó folha para *null*.
   d) Reorganizar os nós à esquerda e à direita do nó folha.
   e) Definir o nó folha como nó raiz da árvore.

3. Quando manipulamos árvores binárias e nos deparamos com a exclusão de nós, a situação pode variar, dependendo das características do nó em questão. Se o nó a ser excluído tiver apenas um filho, o processo de exclusão é mais simples do que em outros casos.

   Considerando a situação de exclusão de nós em árvores binárias, qual é o procedimento para excluir um nó que tem apenas um filho?

   a) Remover o nó filho do nó a ser excluído.
   b) Definir o nó filho como novo nó raiz da árvore.
   c) Alterar o ponteiro do filho do nó a ser excluído para *null*.
   d) Substituir o nó a ser excluído pelo seu único filho, ajustando o ponteiro do pai do nó a ser excluído para apontar para o filho do nó a ser excluído.
   e) Desconectar o nó a ser excluído da árvore, mantendo seu filho conectado aos nós adjacentes.

REFERÊNCIAS

BACKES, A. R. **Algoritmos e estruturas de dados em linguagem C**. Rio de Janeiro: LTC, 2023.

OLIVEIRA, P. M. de, PEREIRA, R. de L. **Estrutura de dados II**. Maringá: Unicesumar, 2019.

PEREIRA, S. do L. **Estruturas de dados em C**: uma abordagem didática. São Paulo: Érica, 2016.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 2010.

CONFIRA SUAS RESPOSTAS

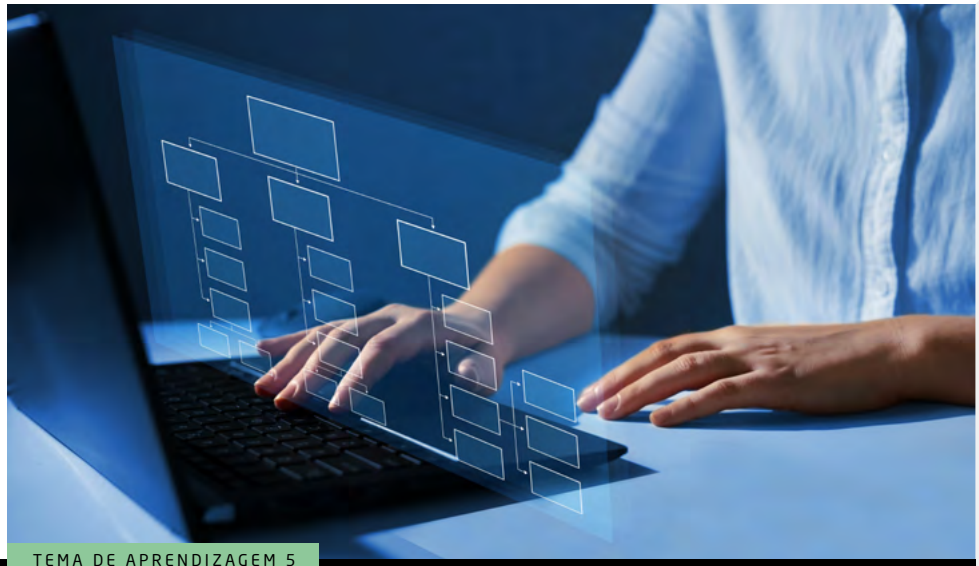
1. Alternativa B.

2. Alternativa C.

3. Alternativa D.

MEU ESPAÇO

MEU ESPAÇO



TEMA DE APRENDIZAGEM 5

ÁRVORES AVL

MINHAS METAS

- Compreender conceitos iniciais de árvores AVL.
- Entender a operação envolvida em árvores AVL.
- Compreender como funciona o processo de adição em AVL.
- Compreender como funciona o processo de remoção em AVL.
- Aprender as diferentes árvores AVL.
- Compreender os passos envolvidos para balanceamento de árvores AVL.
- Verificar, na prática, árvores AVL.

INICIE SUA JORNADA

Imagine que você está desenvolvendo um sistema de gerenciamento de estoque para uma empresa de e-commerce em crescimento. A demanda por produtos está aumentando rapidamente, e a eficiência na busca e manipulação de dados se torna crucial para garantir uma experiência de compra satisfatória para os clientes. Entretanto, conforme o número de produtos armazenados aumenta, você começa a perceber uma queda no desempenho do sistema, especialmente durante as operações de pesquisa e atualização de estoque. Como resolver esse problema de desempenho e garantir que o sistema possa lidar com o aumento da carga de trabalho de forma eficiente?

A resolução do problema de desempenho no sistema de gerenciamento de estoque é fundamental para garantir a competitividade e a sustentabilidade do negócio. Um sistema lento pode resultar em experiências negativas para os clientes, levando a perdas de vendas e reputação. Além disso, a eficiência na manipulação de dados afeta diretamente a produtividade dos funcionários e os custos operacionais da empresa. Portanto, investir na otimização do sistema é essencial para manter a competitividade e o crescimento do negócio.

PLAY NO CONHECIMENTO

Você já ouviu falar de árvores AVL? Que tal se aprofundar nesse conteúdo e entender como acontece as etapas para execução desse tipo de árvore? Neste vídeo, você observará importantes conceitos acerca desse assunto. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

VAMOS RECORDAR?

Que tal recordar as árvores binárias? Esse conteúdo é de grande relevância para identificar o processo de árvores AVL.

Acesse: **https://www.youtube.com/watch?v=b-0h8gqTrlw.**

DESENVOLVA SEU POTENCIAL

COMPREENDENDO AS ÁRVORES AVL

A forma como os elementos são inseridos em uma árvore binária de busca pode fazer com que a busca se torne altamente ineficiente. Considere o conjunto de dados a seguir:

2 - 5 - 10 - 17 - 25 - 32

Se inserirmos tais dados, do primeiro até o último, em uma árvore de busca binária, teremos a árvore estruturada da seguinte forma:

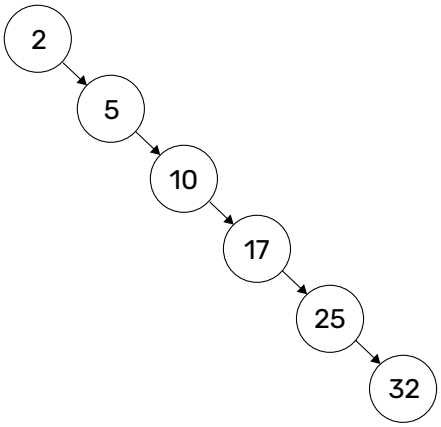


Figura 1 – Árvore de busca binária desbalanceada / Fonte: os autores.

Descrição da Imagem: a figura apresenta nós conectados por arestas, ou linhas. No total, são seis nós conectados em uma linha diagonal. Fim da descrição.

Repare que a estrutura de dados se assemelha muito mais a uma lista linear encadeada, do que com uma árvore binária de busca. Em consequência, operações de busca nessa estrutura de dados serão feitas de maneira sequencial, em vez de aproveitarem-se da capacidade de ignorar subárvores, cortando caminho na busca. Dizemos que a árvore da Figura 1 encontra-se desbalanceada.

APROFUNDANDO

As árvores AVL são uma forma de árvore binária de busca balanceada. Elas são projetadas para garantir que a altura da árvore permaneça equilibrada, o que ajuda a manter um desempenho de busca eficiente. O nome AVL vem dos seus inventores, Adelson-Velsky e Landis (Backes, 2023).

Em uma árvore AVL, a diferença entre as alturas das subárvores esquerda e direita (conhecida como fator de balanceamento), de cada nó, é mantida dentro de um limite específico, geralmente -1, 0 ou 1. Se após uma operação (inserção ou remoção) essa propriedade não for mais válida, a árvore é reequilibrada por meio de rotações.

A partir daí, surge o conceito de balanceamento. Dizemos que uma árvore balanceada tende a manter sua altura tão pequena quanto possível, à medida em que são realizadas novas inserções ou remoções de dados. Considerando o mesmo conjunto de dados que vimos anteriormente, poderíamos tentar balancear a árvore binária de busca, resultando em algo parecido com o seguinte (Oliveira; Pereira, 2019):

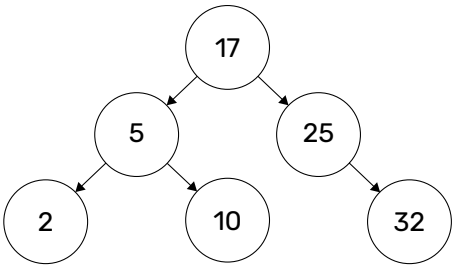


Figura 1 – Árvore de busca binária desbalanceada / Fonte: os autores.

Descrição da Imagem: a figura apresenta nós conectados por arestas, ou linhas. No total, são seis nós conectados em uma linha diagonal. Fim da descrição.

O conceito de balanceamento está relacionado à altura das subárvores que compõem uma árvore binária. É importante lembrar que a altura (ou profundidade) de uma subárvore é igual ao número de nós visitados desde a raiz até o nó folha mais distante (uma subárvore vazia possui altura -1, por definição). Dizemos que um nó está balanceado caso o valor absoluto da diferença entre as alturas das subárvores esquerda e direita seja menor ou igual a 1.

Chamaremos de fator de balanceamento (Fb) o resultado da diferença entre a altura da subárvore esquerda (He) de um nó pela altura da subárvore direita (Hd) do mesmo nó, de acordo com a seguinte fórmula:

$$Fb = He - Hd$$

Vamos a alguns exemplos. A árvore da Figura 3 encontra-se balanceada. Nessa estrutura, ilustramos a altura (H) de cada subárvore na porção superior esquerda de cada nó, e o respectivo fator de balanceamento (Fb) no canto superior direito de cada nó.

Ademais, as raízes de subárvores vazias estão destacadas como quadrados pontilhados. Repare como cada fator de balanceamento (destacados ao lado superior direito de cada nó) é o resultado do valor da altura da subárvore esquerda menos a altura da subárvore direita.

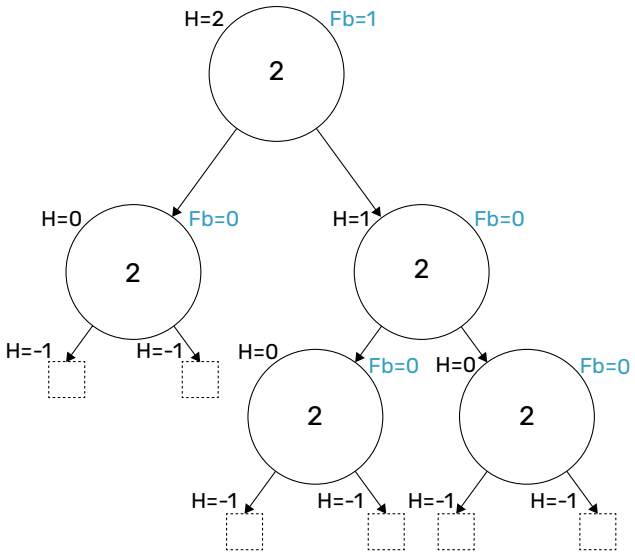


Figura 3 - Fatores de balanceamento com todos os nós balanceados
Fonte: os autores.

Descrição da Imagem: a figura apresenta uma árvore binária conectada por cinco nós. A conexão acontece a partir de arestas, ou linhas. Alguns nós não possuem valores, representando as diversas possibilidades de inclusão de nós filhos. Todavia, existem situações nas quais uma árvore de busca possui nós desbalanceados. Fim da descrição.

Imagine, por exemplo, que adicionamos um nó cujo valor é igual a 32. Seguindo as regras de inserção para uma árvore binária de busca, tal nó seria adicionado como filho à direita do nó de valor igual a 25. Com isso, temos uma nova árvore, como ilustra a Figura 4 a seguir.

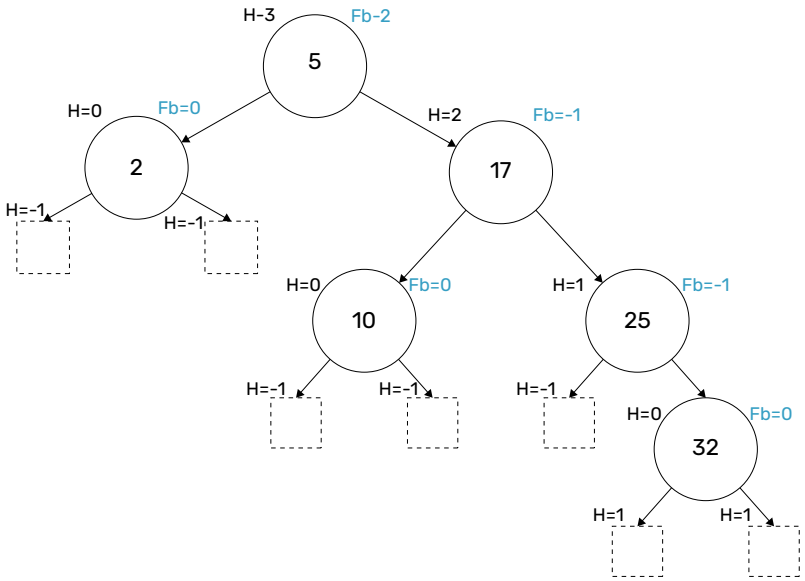


Figura 4 – Fatores de balanceamento com um nó desbalanceado / Fonte: os autores.

Descrição da Imagem: a figura apresenta uma árvore binária conectada por seis nós. A conexão acontece a partir de arestas, ou linhas. Alguns nós não possuem valores, representando as diversas possibilidades de inclusão de nós filhos. Fim da descrição.

Analisando a figura anterior, notamos que a inserção do nó de valor 32 fez com que a raiz da árvore perdesse seu balanceamento. Note que a altura da subárvore esquerda, em relação à raiz, é igual a 0, porém, a altura da subárvore direita é igual a 2. Com isso, o fator de balanceamento da raiz é igual a -2 que, em valor absoluto, é maior do que 1. Assim, pela definição, dizemos que o nó de valor igual a 5 encontra-se desbalanceado.

Para resolver o problema do desbalanceamento de árvores binárias de busca, os pesquisadores Adelson-Velskii e Landis, em 1962, criaram um algoritmo que leva as iniciais de seus nomes. As árvores AVL são, nesse sentido, árvores nas quais todos os nós encontram-se balanceados. Para cada nó, a diferença entre as alturas de suas subárvores não pode ser igual ou superior a 2, em valor absoluto (Pereira, 2016).

Como vimos, uma árvore balanceada pode perder essa característica quando um novo elemento é inserido. Isso também pode ocorrer quando um elemento é removido da árvore binária. Assim, quando ocorrem operações de inserção ou remoção em árvores AVL, recalcula-se os fatores de balanceamento de cada nó para, assim, poder executar rotações nos nós problemáticos, na tentativa de restabelecer seu balanceamento. Existem, basicamente, quatro categorias de rotações: duas simples e duas duplas.

Dessa forma, podemos observar que, no exemplo da Figura 8, foi realizada uma inserção na subárvore direita do filho direito em relação ao nó desbalanceado.

Em outras palavras, o nó 32 foi inserido na subárvore direita do nó 17, que é filho à direita do nó 5. Para tornar a árvore balanceada, precisamos realizar uma rotação, conforme o ilustrado na Figura 5 a seguir.

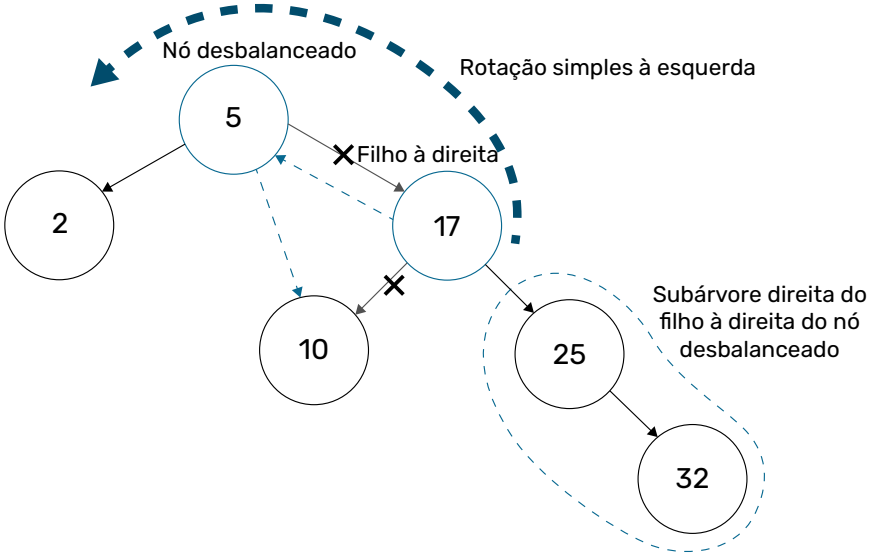


Figura 5 – Corrigindo uma inserção na subárvore direita do filho à direita com rotação simples à esquerda
Fonte: os autores.

Descrição da Imagem: a figura apresenta uma árvore binária conectada por seis nós. A conexão acontece a partir de arestas, ou linhas. A imagem ilustra a possibilidade de balanceamento de árvores, identificando o processo de rotação de árvores. Fim da descrição.

APROFUNDANDO

O tempo de execução das operações básicas (inserção, remoção e busca) em uma árvore AVL é O (log n), em que n é o número de nós na árvore. Isso torna as árvores AVL uma escolha popular quando é necessário manter uma grande quantidade de dados classificados e garantir tempos de pesquisa eficientes.

Para realizar a rotação à esquerda, precisamos puxar o nó 17 para cima, fazendo com que o nó 5 passe a ser o filho à esquerda do nó 17. Além disso, a subárvore com raiz no nó 10 é deserdada pelo nó 17, e o nó 5 a adota, fazendo com que o nó 10 passe a ser o filho à direita do nó 5, deixando de ser o filho à esquerda do nó 17. Dessa forma, após realizada a rotação, devemos recalcular a altura de cada subárvore, bem como os fatores de balanceamento de cada nó. Nesse caso, a árvore balanceada resultante pode ser visualizada na Figura 6 a seguir.

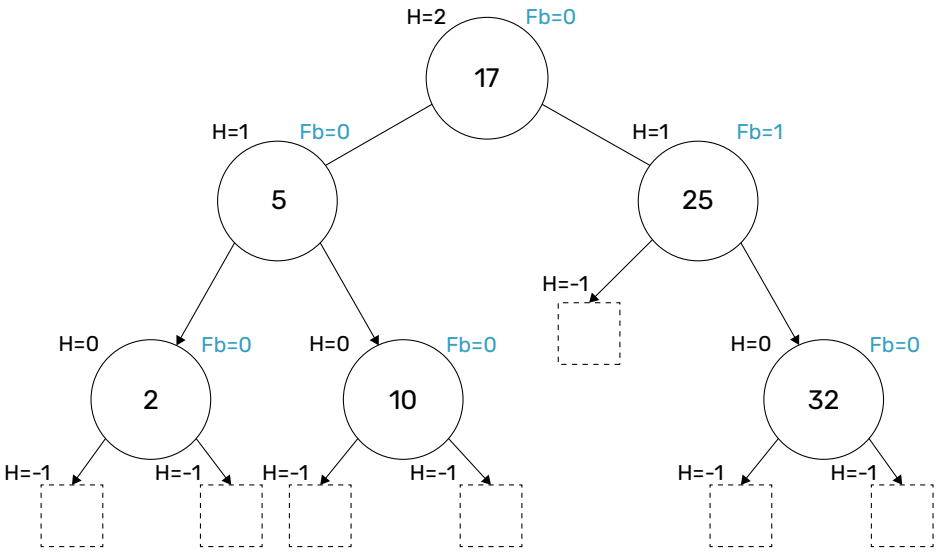


Figura 6 - Árvore AVL balanceada após uma inserção / Fonte: os autores.

Descrição da Imagem: a figura apresenta uma árvore binária conectada por seis nós. A conexão acontece a partir de arestas, ou linhas. Alguns nós não possuem valores, representando as diversas possibilidades de inclusão de nós filhos. Fim da descrição.

A partir dessa nova árvore balanceada, adicionaremos um nó de valor igual a 1. Para isso, devemos seguir a regra de inserção em árvore de busca binária. Portanto, temos de percorrer da raiz 17, em direção a uma folha, até poder inserir o 1 à esquerda do nó 2. Dessa forma, temos uma nova árvore, com novas alturas e fatores de balanceamento, como podemos observar na Figura 7 a seguir. Repare que a árvore não está desbalanceada em nó algum.

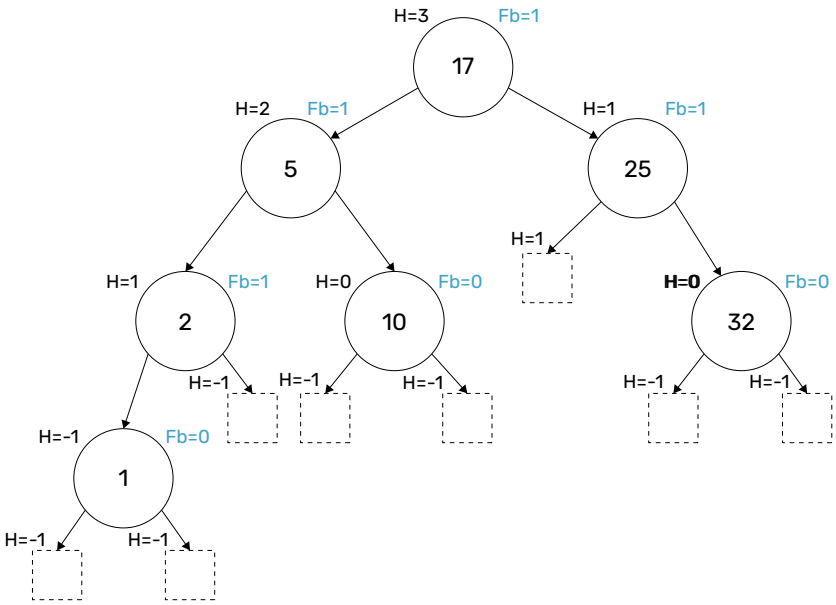


Figura 7 – Inserindo o nó 1 / Fonte: os autores.

Descrição da Imagem: a figura mostra uma árvore binária conectada por sete nós. A conexão acontece a partir de arestas, ou linhas. Alguns nós não possuem valores, representando as diversas possibilidades de inclusão de nós filhos. Fim da descrição.

Sempre que inserimos um nó, devemos checar se todos os nós percorridos, desde a raiz até o nó folha, estão balanceados. Como podemos observar na Figura 7 anterior, os nós percorridos foram, nesta ordem: 17, 5 e 2. É possível reparar que, após a inserção, foram recalculados os fatores de balanceamento Fb de tais nós e nenhum deles se tornou desbalanceado. Repare, ainda, que somente os nós 17, 5 e 2 tiveram seus fatores de balanceamento alterados, isto é, reforça o fato de que apenas os nós percorridos devem ser checados à procura de desbalanceamentos.

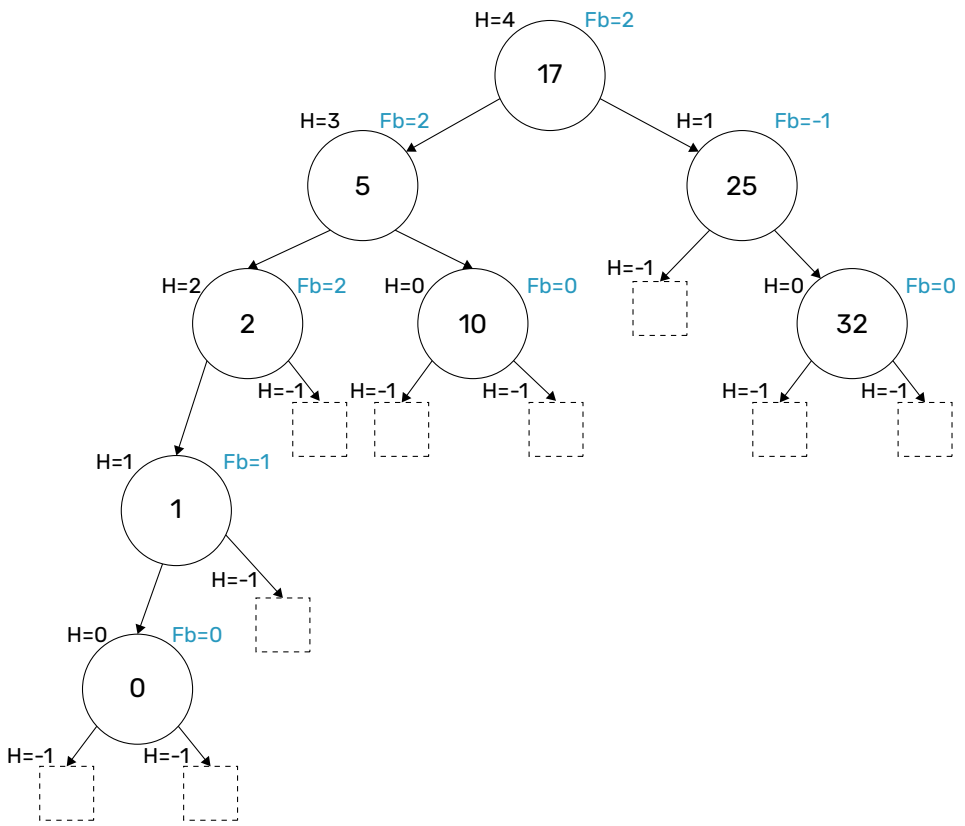


Figura 8 – Inserindo o nó 0 / Fonte: os autores.

Descrição da Imagem: a figura mostra uma árvore binária conectada por oito nós. A conexão acontece a partir de arestas, ou linhas. Alguns nós não possuem valores, representando as diversas possibilidades de inclusão de nós filhos. Fim da descrição.

Note, agora, que, na Figura 8 anterior, temos a inserção do nó de valor igual a 0. Para adicionar o nó 0, tivemos de percorrer os nós 17, 5, 2 e 1 para, finalmente, posicionar o 0 à esquerda do nó 1. Ao recalcular os fatores de balanceamento, chegamos à conclusão de que os nós 2, 5 e 17 estão desbalanceados. A partir dessas informações, temos condições de tentar consertar nossa árvore, de baixo para cima. Novamente, devemos dar atenção ao fato de que apenas os nós percorridos durante a inserção tiveram chance de se tornar desbalanceados, pois somente suas subárvores tiveram as respectivas alturas alteradas após a inserção.

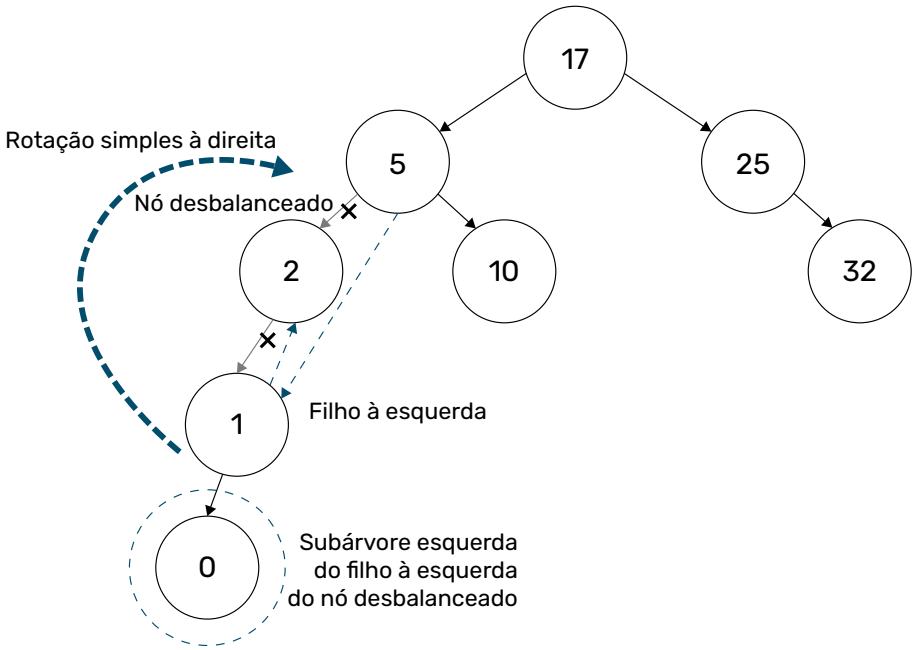


Figura 9 – Corrigindo uma inserção na subárvore esquerda do filho à esquerda com rotação simples à direita
Fonte: os autores.

Descrição da Imagem: a figura demonstra uma árvore binária conectada por oito nós. A conexão acontece a partir de arestas, ou linhas. A figura ilustra a possibilidade de balanceamento de árvores, identificando o processo de rotação de árvores. Fim da descrição.

De maneira simétrica à situação que levou à rotação feita anteriormente, temos o caso oposto. Agora, realizamos a inserção de um nó na subárvore esquerda do filho à esquerda do nó desbalanceado. Para resolver esse desbalanceamento, devemos realizar uma rotação simples para a direita, fazendo com que o nó 5 passe a ser o pai do nó 1 que, por sua vez, passa a ser o filho à esquerda do nó 5. O nó 2 deixa de ser filho do nó 5 e passa a ser o filho à direita do nó 1. Caso o nó 1 tenha alguma subárvore à direita, a raiz dessa subárvore passa a ser filha à esquerda do nó 2 – o qual adota o filho órfão do nó 1. O resultado dessa rotação pode ser observado na Figura 10 a seguir.

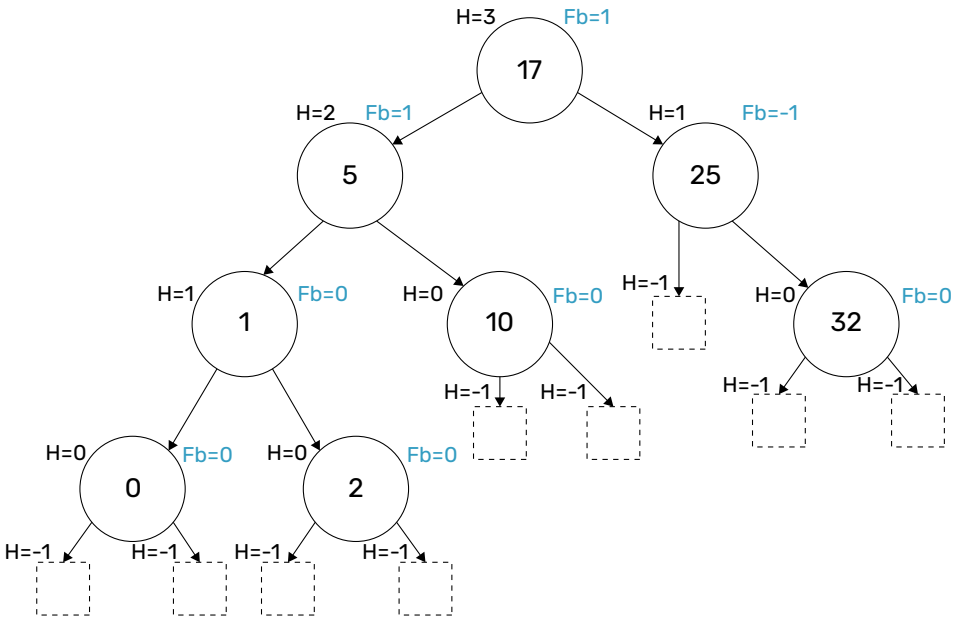


Figura 10 – Balanceamento de árvore AVL após rotação simples à direita
Fonte: os autores.

Descrição da Imagem: a figura demonstra a árvore binária conectada por oito nós. A conexão acontece a partir de arestas, ou linhas. Alguns nós não possuem valores, representando as diversas possibilidades de inclusão de nós filhos. Fim da descrição.

A Figura 10 anterior demonstra que, realizando uma rotação simples no primeiro nó desbalanceado, partindo do nó recém-inserido em direção à raiz, balanceou-se toda a árvore. Todavia, em situações nas quais os nós ascendentes continuam desbalanceados após uma rotação, é necessário continuar corrigindo a árvore até que todos os nós da folha, até a raiz, sejam balanceados.

Considere outra árvore, agora, em que os nós 17, 32 e 25 foram inseridos na árvore de busca binária, na respectiva ordem. Com isso, temos a raiz 17 se tornando desbalanceada, como podemos ver na Figura 11 a seguir. Contudo, a quebra no balanço foi realizada por meio de uma inserção na subárvore esquerda do filho direito em relação ao nó desbalanceado, ou seja, o número 25 foi inserido à esquerda do nó 32 que, por sua vez, está à direita do nó 17.

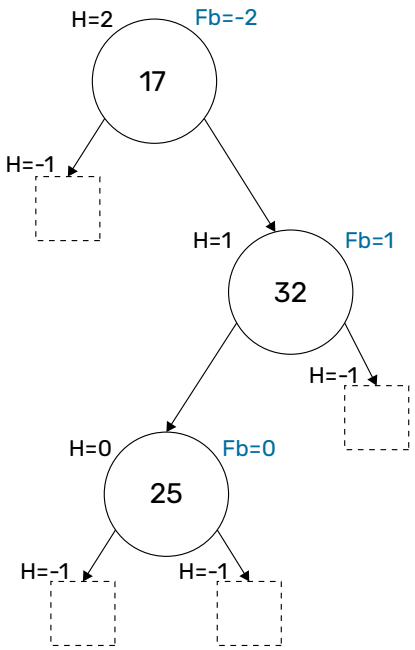


Figura 11 – Inserção na subárvore esquerda do filho à direita do nó desbalanceado
Fonte: os autores.

Descrição da Imagem: a figura demonstra uma árvore binária conectada por três nós. A conexão acontece a partir de arestas, ou linhas. Alguns nós não possuem valores, representando as diversas possibilidades de inclusão de nós filhos. Fim da descrição.

Em uma situação como essas, uma única rotação simples em torno do nó desbalanceado não resolve o problema. Isso ocorre, pois, ao tentar realizar uma única rotação à esquerda, continuaríamos tendo o nó raiz 17 desbalanceado. Nessas situações, é necessário empregar a rotação dupla direita-esquerda, que é feita em duas etapas. Nesse caso, primeiramente, rotaciona-se, à direita, a subárvore com raiz em 25, em direção ao nó 32; num segundo momento, rotaciona-se, à esquerda, a subárvore com raiz em 25, em direção ao nó 17, como podemos ver na Figura 12 a seguir (Szwarcfiter; Markenzon, 2010).

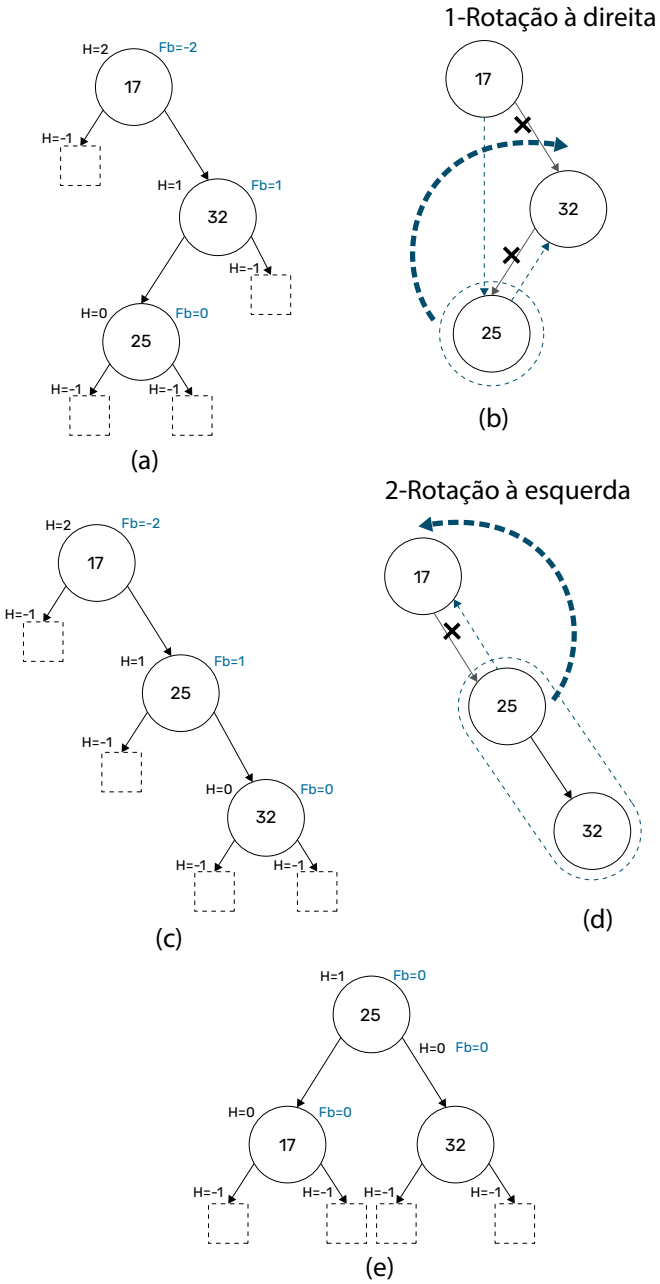


Figura 12 – Corrigindo uma inserção na subárvore esquerda do filho à direita com rotação dupla esquerda-direita
Fonte: os autores.

Descrição da Imagem: a figura demonstra árvores binárias conectadas por três nós. A conexão acontece a partir de arestas, ou linhas. A imagem ilustra como acontece o processo de rotação de árvores para realizar o balanceamento delas. Fim da descrição.

Em suma, uma árvore AVL é uma árvore binária de busca na qual o fator de balanceamento de cada um de seus nós não pode ser maior que 1 ou menor que -1. Caso o fator de balanceamento de um ou mais nós seja maior ou igual a 2, em valores absolutos, é necessário analisar a ordem das inserções para aplicar a rotação mais adequada para corrigir o balanceamento de cada um dos nós. Isso faz com que buscas em uma árvore AVL sejam mais eficientes.

EM FOCO

Acesse seu ambiente virtual de aprendizagem e confira a aula referente a este tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

NOVOS DESAFIOS

Com relação ao problema apresentado anteriormente, os estudantes podem experimentar a resolução dessa questão, implementando uma árvore AVL para o armazenamento e gerenciamento de dados de estoque. Ao comparar o desempenho do sistema antes e depois da implementação da árvore AVL, eles podem identificar como a estrutura de dados balanceada contribui para melhorar a eficiência nas operações de pesquisa e atualização. Além disso, podem explorar diferentes técnicas de otimização, como cache de dados e paralelização de operações, para maximizar o desempenho do sistema em diferentes cenários de uso.

A experiência de resolver o problema de desempenho no sistema de gerenciamento de estoque não apenas demonstra a importância da escolha adequada de estruturas de dados, mas também destaca a necessidade contínua de adaptação e melhoria em ambientes de negócios dinâmicos. Além disso, incentiva os estudantes a considerar não apenas a eficiência técnica, mas também os impactos econômicos e operacionais de suas decisões de projeto. Essa reflexão os prepara para enfrentar desafios semelhantes no mundo real e a buscar soluções inovadoras e eficazes para problemas complexos de engenharia de software.

1. Imagine que você é um desenvolvedor de software, trabalhando em um projeto de gestão de uma biblioteca digital. Você está projetando a estrutura de dados para armazenar os livros disponíveis na biblioteca de forma eficiente. Considerando a necessidade de realizar operações de busca e inserção de livros de forma rápida e eficiente, você decide implementar uma árvore AVL para organizar os dados (Backes, 2023).

   Qual é a principal característica de uma árvore AVL que a torna adequada para o armazenamento eficiente de dados em uma biblioteca digital?

   a) A capacidade de armazenar uma grande quantidade de dados sem afetar o desempenho do sistema.
   b) A estrutura hierárquica que facilita a organização e busca eficiente de dados.
   c) A garantia de que a altura da árvore permaneça equilibrada, mantendo operações de busca em O(log n).
   d) A capacidade de realizar operações de inserção e remoção de dados em tempo constante.
   e) A flexibilidade para lidar com diferentes tipos de dados, como textos, imagens e vídeos.

2. Você está desenvolvendo um sistema de gerenciamento de contatos para um aplicativo de mensagens. Para garantir uma busca rápida e eficiente dos dados dos usuários, você decide implementar uma estrutura baseada em árvores AVL (Backes, 2023).

   Qual é a complexidade de tempo para realizar uma operação de busca em uma árvore AVL que contém n elementos?

   a) O(1).
   b) O(n).
   c) O(log n).
   d) O(n log n).
   e) $O(n^2)$.

3. **Você está trabalhando em um projeto de desenvolvimento de software para uma empresa de logística, em que a eficiência na gestão de rotas e entregas é crucial. Para otimizar o processo de planejamento de rotas, você decide implementar uma estrutura de dados baseada em árvores AVL (Backes, 2023).**

   Com base nas informações apresentadas, avalie as asserções, a seguir, e a relação proposta entre elas:

   I - As árvores AVL são uma forma de árvore de busca binária balanceada.

   PORQUE

   II - Uma árvore AVL garante que a altura das subárvores esquerda e direita de cada nó seja mantida equilibrada, mantendo, assim, a complexidade de busca em O(log n).

   A respeito dessas asserções, assinale a opção correta:

   a) As asserções I e II são verdadeiras, e a II é uma justificativa correta da I.
   b) As asserções I e II são verdadeiras, mas a II não é uma justificativa correta da I.
   c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
   d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
   e) As asserções I e II são falsas.

REFERÊNCIAS

BACKES, A. R. **Algoritmos e estruturas de dados em linguagem C**. Rio de Janeiro: LTC, 2023.

OLIVEIRA, P. M. de, PEREIRA, R. de L. **Estrutura de dados II**. Maringá: Unicesumar, 2019.

PEREIRA, S. do L. **Estruturas de dados em C**: uma abordagem didática. São Paulo: Érica, 2016.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 2010.

CONFIRA SUAS RESPOSTAS

1. Alternativa C.

   Uma árvore AVL é uma estrutura de busca binária balanceada, o que significa que a diferença entre as alturas das subárvores esquerda e direita de cada nó é mantida dentro de um limite específico. Isso garante que a árvore permaneça equilibrada e que as operações de busca tenham complexidade de tempo O(log n), em que n é o número de elementos na árvore.
   A alternativa A está incorreta, pois embora as árvores AVL sejam eficientes em termos de desempenho, sua principal vantagem está na garantia de altura balanceada, não na capacidade de armazenar grandes quantidades de dados.
   A alternativa B está incorreta, pois a estrutura hierárquica, que facilita a organização e busca eficiente de dados, descreve uma característica geral das árvores de busca binária, não especificamente das árvores AVL. Embora as árvores AVL sejam uma forma de árvore de busca binária, sua principal vantagem está na manutenção da altura balanceada, não apenas na estrutura hierárquica.
   A alternativa D está incorreta, pois a capacidade de realizar operações de inserção e remoção de dados em tempo constante não é uma característica das árvores AVL. Embora tais estruturas sejam eficientes, em termos de desempenho, para operações de busca O(log n), as operações de inserção e remoção podem exigir rotações e ajustes na estrutura da árvore, o que resulta em complexidade de tempo O(log n).
   A alternativa E está incorreta, pois a flexibilidade para lidar com diferentes tipos de dados, como textos, imagens e vídeos, não é uma característica específica das árvores AVL. Tal capacidade de lidar com diferentes tipos de dados depende da implementação específica do software e não da estrutura da árvore em si.

2. Alternativa C.

   Em uma árvore AVL, as operações de busca têm complexidade de tempo O(log n), em que n é o número de elementos na árvore. Isso ocorre, pois a árvore AVL é uma estrutura de busca binária balanceada, o que significa que a altura da árvore é mantida equilibrada, garantindo que a busca possa ser feita de forma eficiente, percorrendo apenas uma fração do total de elementos.
   A alternativa A está incorreta, pois embora algumas estruturas de dados, como tabelas de dispersão (*hash tables*), possam oferecer uma complexidade de tempo de O(1) para operações de busca em casos ideais, uma árvore AVL não possui essa característica. A complexidade de tempo de uma árvore AVL para busca é O(log n).
   A alternativa B está incorreta, pois uma complexidade de tempo de O(n) ocorre geralmente em algoritmos que precisam percorrer todos os elementos da estrutura de dados para encontrar o resultado desejado. No caso de uma árvore AVL, a complexidade de busca é muito mais eficiente, sendo O(log n).

A alternativa D está incorreta, pois tal complexidade de tempo é comum em algoritmos de ordenação eficientes, como o *Merge Sort* ou o *Quick Sort*. Em uma árvore AVL, a complexidade de tempo para operações de busca é mais eficiente, sendo O(log n).
A alternativa E está incorreta, pois uma complexidade de tempo de $O(n^2)$ é comum em algoritmos de ordenação menos eficientes, como *Bubble Sort* ou *Selection Sort*, em que o tempo aumenta quadraticamente com o aumento do número de elementos. Em uma árvore AVL, a complexidade de tempo para operações de busca é muito mais eficiente, sendo O(log n).

3. Alternativa A.

Ambas as asserções estão corretas, e a II é uma justificativa correta da I.
A asserção I está correta, pois as árvores AVL são de fato uma forma de árvore de busca binária balanceada. Isso significa que a estrutura de uma árvore AVL é organizada de tal forma que a diferença entre as alturas das subárvores esquerda e direita de cada nó é mantida dentro de um limite específico, garantindo um equilíbrio na altura da árvore.
A asserção II está correta, pois uma das principais propriedades das árvores AVL é que elas garantem que a altura das subárvores esquerda e direita de cada nó seja mantida equilibrada. Isso significa que, após qualquer operação de inserção ou remoção, a diferença de altura entre as subárvores de um nó (fator de balanceamento) é no máximo 1. Como resultado desse balanceamento, a complexidade de busca em uma árvore AVL é O(log n), em que n é o número de elementos na árvore.
Portanto, a asserção I está correta ao afirmar que as árvores AVL são uma forma de árvore de busca binária balanceada, e a asserção II está correta ao afirmar que uma árvore AVL garante que a altura das subárvores esquerda e direita de cada nó seja mantida equilibrada, o que, por sua vez, mantém a complexidade de busca em O(log n). Além disso, a asserção II justifica a asserção I, pois a garantia de balanceamento da altura é uma característica fundamental das árvores AVL que as torna uma forma eficiente de árvore de busca binária balanceada.

MEU ESPAÇO

MEU ESPAÇO

Unicesumar

unidade 4

TEMA DE APRENDIZAGEM 6

ÁRVORES B

MINHAS METAS

- Compreender conceitos iniciais de árvores B.
- Entender a operação envolvida em árvores B.
- Aprender como funciona o processo de adição em B.
- Saber como funciona o processo de remoção em B.
- Estudar árvores diferentes.
- Examinar os passos envolvidos para balanceamento de árvores B.
- Verificar, na prática, as árvores B.

INICIE SUA JORNADA

Imagine que você é um desenvolvedor de software recém-formado e está trabalhando em uma empresa de tecnologia. Sua equipe recebeu a tarefa de otimizar a busca de informações em um grande banco de dados, que está se tornando, cada vez mais, lento e ineficiente, à medida que a quantidade de elementos cresce. A solução atual, baseada em uma simples estrutura de lista, não é mais suficiente para lidar com o aumento exponencial das informações. Como você pode garantir que as buscas sejam realizadas de maneira eficiente e em tempo hábil, mesmo com o crescimento contínuo do banco de dados?

Resolver essa problemática é importante não apenas para o sucesso do projeto em que você está trabalhando, mas também para sua carreira profissional. A eficiência na busca de dados é fundamental para qualquer aplicação que lida com grandes volumes de informação. Se a busca for lenta, isso pode causar a insatisfação dos usuários, a perda de clientes e até mesmo o comprometimento da reputação da empresa. Portanto, aprender e aplicar estruturas de dados eficientes, como a árvore B, pode ter um impacto significativo na sua capacidade de desenvolver soluções robustas e escaláveis.

PLAY NO CONHECIMENTO

Você já ouviu falar em árvores B? Que tal se aprofundar nesse conteúdo e entender como acontecem as etapas para execução desse tipo de estrutura? Neste vídeo, você verá importantes conceitos acerca desse assunto. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

VAMOS RECORDAR?

Que tal recordar as noções de árvores binárias? O conteúdo possui grande relevância para a identificação do processo de árvores AVL.

Acesse: **https://www.youtube.com/watch?v=VmKkAQtnjsM**.

DESENVOLVA SEU POTENCIAL

COMPREENDENDO AS ÁRVORES B

Uma árvore B é uma estrutura amplamente utilizada em bancos de dados e sistemas de arquivos para organizar e armazenar elementos de forma eficiente. Ela é projetada para proporcionar rápida inserção, remoção e busca, especialmente, em grandes conjuntos de informações (Backes, 2023; Szwarcfiter; Markenzon, 2010).

Aqui, está uma introdução detalhada das árvores B:

Estrutura básica

Uma árvore B consiste em um conjunto de nós interligados. Cada nó é composto por duas partes principais: chaves e ponteiros.

- Chaves: são os valores armazenados na árvore, os quais estão ordenados dentro de cada nó.
- Ponteiros: são as referências para outros nós na árvore. Eles guiam a busca na árvore, permitindo navegar de um nó para outro.

Características principais

- Ordem da árvore (m): define o número máximo de filhos que um nó pode ter. Cada nó, exceto a raiz, deve ter pelo menos (m/2) chaves e no máximo m chaves. Quanto maior o valor de m, mais chaves podem ser armazenadas em cada nó, o que resulta em árvores mais largas e rasas.
- Raiz: é o nó no topo da árvore. Inicialmente, quando a árvore está vazia, a raiz é criada. À medida que a árvore cresce, a raiz pode se dividir para acomodar novas chaves.
- Nós internos: são os nós que não são folhas. Eles contêm chaves e ponteiros para os filhos.
- Nós-Folha: são os nós na extremidade da árvore. Eles contêm apenas chaves, mas não têm ponteiros para outros nós.

Operações principais

- Inserção: para inserir um novo valor na árvore, comece na raiz e desça até o nó-folha apropriado. Se o nó estiver cheio, ele é dividido e a chave mediana é promovida para o nível superior. Esse processo pode se repetir recursivamente até a raiz.
- Remoção: para remover uma chave, encontre o nó-folha que a contém e remova-a. Se o número de chaves no nó ficar abaixo do mínimo permitido, pode ser necessário redistribuir as chaves entre os nós ou mesclar dois nós vizinhos.
- Busca: começando na raiz, compare a chave de busca com as chaves no nó atual e siga o ponteiro apropriado para o próximo nível da árvore. Continue esse processo até encontrar a chave desejada ou chegar a um nó-folha.

Vantagens das árvores B

- Eficiência: as operações de inserção, remoção e busca têm complexidade de tempo O (log n), em que n é o número de chaves na árvore.
- Balanceamento automático: a estrutura da árvore é automaticamente ajustada durante as operações de inserção e remoção para manter a árvore balanceada.
- Uso eficiente de memória: a árvore B é projetada para armazenar grandes quantidades de dados em disco, minimizando o número de acessos necessários para recuperar informações (Oliveira; Pereira, 2019).

Aplicações

- Bancos de dados: árvores B são amplamente utilizadas em sistemas de gerenciamento de banco de dados para indexação de elementos e consultas eficientes.
- Sistemas de arquivos: muitos sistemas de arquivos, como o ext4 no Linux, usam árvores B para organizar e indexar blocos de dados em discos rígidos.
- Sistemas de Informação Geográfica (GIS): são utilizados para armazenar e consultar dados espaciais, como mapas e coordenadas geográficas.

APROFUNDANDO

Um dos principais usos das árvores B, em bancos de dados, é a indexação. As árvores B são eficientes para organizar e acessar grandes conjuntos de dados de forma rápida. Elas são frequentemente usadas em índices de chaves primárias e secundárias para otimizar consultas de busca. Graças à sua estrutura balanceada e organizada, as árvores B permitem uma busca rápida e eficiente de dados. Isso é fundamental em bases, nas quais a eficiência das consultas pode afetar diretamente o desempenho do sistema.

Vamos criar um exemplo simples de uma árvore B para armazenar números inteiros. Nele, consideraremos uma árvore B de ordem 3, o que significa que cada nó pode conter até três chaves (Pereira, 2016).

Passo 1: criando a raiz

Começamos com uma árvore vazia. Quando inserimos o primeiro elemento (neste caso, o número 5), ele se torna a raiz da árvore. Como é o primeiro elemento, não há necessidade de dividir ou rearranjar nada, pois a raiz pode conter até três chaves (de acordo com a ordem da árvore).

Passo 2: inserindo mais elementos

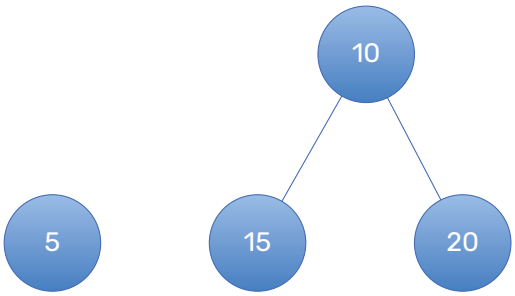
Vamos inserir mais alguns números na árvore. O número 10 é colocado como o segundo elemento. Como a raiz ainda possui espaço para mais chaves, o número 10 é simplesmente adicionado a ela.

Passo 3: divisão da raiz

Se tentarmos, no entanto, inserir um novo número, digamos 15, na árvore, a raiz fica cheia e precisa ser dividida. Durante a colocação do número 15, a raiz é dividida em dois nós, criando uma nova com apenas um elemento, que é a chave mediana (no caso, 10). O número 15 é, então, disposto no nó direito.

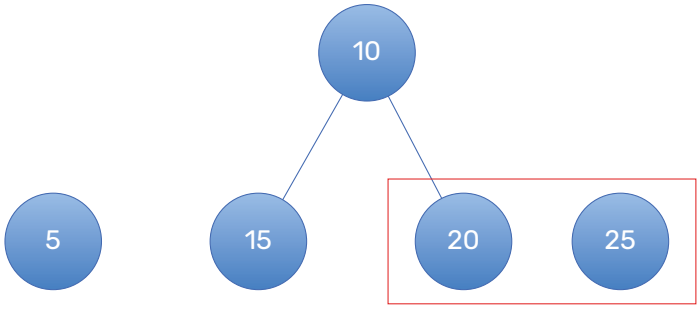
Passo 4: crescimento da árvore

Vamos continuar inserindo mais algarismos para observar o crescimento da árvore. O número 20 é colocado. Como o nó direito da raiz ainda possui espaço, ele é simplesmente adicionado lá.

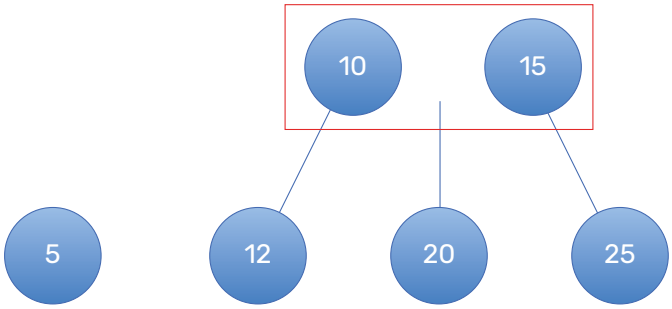


Passo 5: mais níveis envolvidos na árvore

Continuando a inserção, a árvore cresce em altura. O número 25 é colocado. Como o nó direito da raiz está cheio, ele é dividido e a chave mediana (20) é promovida para o nível superior.



O número 12 é inserido na árvore. Como o nó direito da raiz possui espaço, ele é adicionado lá. Entretanto, como os elementos no nó direito ultrapassam a capacidade máxima, ele é dividido e a chave mediana (15) é promovida para o nível superior.

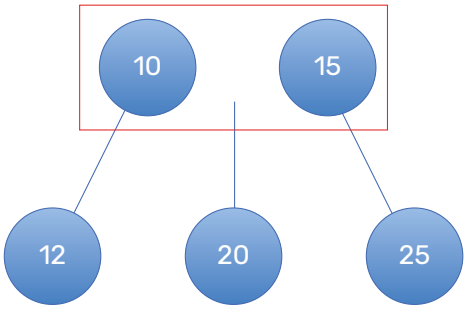


Esse é um exemplo detalhado de como uma árvore B cresce e se ajusta à medida que novos elementos são inseridos. Ela mantém sua estrutura balanceada, garantindo operações eficientes de busca, inserção e remoção.

Com relação ao processo de remoção, vejamos o passo a passo a seguir:

Passo 1: removendo uma chave de um nó-folha

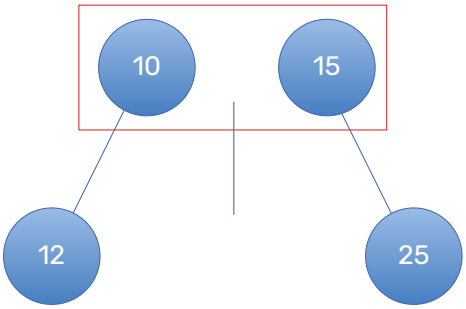
Começamos removendo a Chave 5 da árvore.



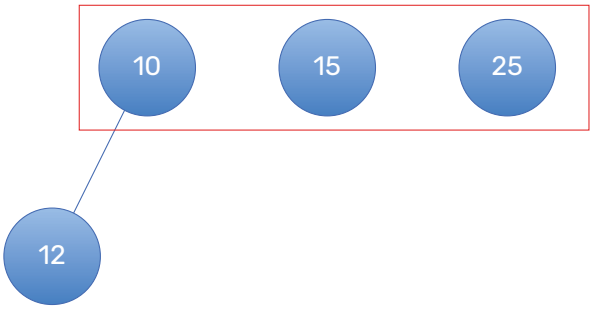
Quando removemos a Chave 5, ela está em um nó-folha, então, podemos simplesmente removê-la.

Passo 2: removendo uma chave de um nó interno

Removeremos, agora, a Chave 20 da árvore.



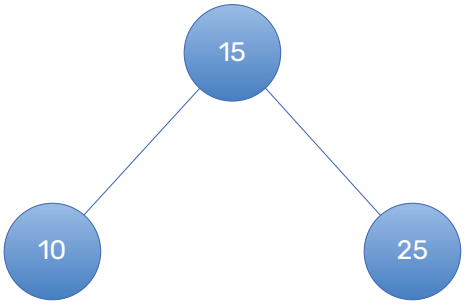
Quando removemos a Chave 20, como ela está em um nó interno, precisamos encontrar o seu sucessor, que é o menor valor no nó-filho direito de 20. O sucessor é 25. Substituímos a Chave 20 pelo seu sucessor e removemos 25 do nó-folha.



Passo 3: removendo uma chave que necessita de redistribuição

Removemos a Chave 12 da árvore.

Quando removemos a Chave 12, deixamos o nó sem chaves suficientes (menos que a metade do máximo). Nesse caso, precisamos redistribuir chaves de um nó vizinho. Como os nós vizinhos não têm chaves suficientes para doar, fundimos os Nós 10, 15 e 25 para formar um único nó.



A árvore, agora, está ajustada e a Chave 12 foi removida com sucesso, mantendo a estrutura balanceada e eficiente para futuras operações de busca e inserção. Essa prática reforça a importância das estruturas de dados na manutenção de sistemas de alta performance e destaca a relevância da árvore B no desenvolvimento de soluções escaláveis e eficazes.

EM FOCO

Acesse seu ambiente virtual de aprendizagem e confira a aula referente a este tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

NOVOS DESAFIOS

Considerando a situação-problema apresentada, inicialmente, neste tema de aprendizagem, você pode implementar uma árvore B em um projeto experimental. Suponhamos que você trabalhe em um sistema de gerenciamento de biblioteca. Substituir a estrutura de dados atual que armazena os livros por uma árvore B, permitirá buscas mais rápidas e eficientes, mesmo à medida que a quantidade de livros aumenta. Ao experimentar essa implementação, você poderá observar, diretamente, como a estrutura de dados melhora o desempenho do sistema e facilita a recuperação de informações.

Ao final desse processo, é essencial refletir acerca dos conhecimentos adquiridos e como eles se aplicam a diferentes contextos. Pergunte-se: como a árvore B pode ser usada em outras áreas da tecnologia da informação? Quais são as limitações dessa estrutura de dados e como você pode superá-las? Como esse conhecimento específico pode contribuir para o seu desenvolvimento profissional e para a inovação na sua área de atuação?

Essas reflexões são fundamentais para consolidar seu aprendizado e explorar novas oportunidades de aplicação prática do que foi aprendido. Considere como a árvore B pode ser integrada em diferentes setores, como sistemas financeiros, motores de busca ou qualquer aplicação que requeira acesso rápido e eficiente a grandes volumes de dados. Identificar as limitações, como a complexidade na implementação e a necessidade de manutenção da árvore balanceada, é igualmente importante para desenvolver estratégias eficazes para superá-las.

Por meio dessa jornada, você não apenas resolverá um problema técnico específico, mas também desenvolverá habilidades que lhe serão valiosas ao longo de toda a sua carreira. A capacidade de aplicar conceitos teóricos a situações práticas, identificar e superar desafios, e inovar continuamente em sua área de atuação, são competências que lhe permitirão destacar-se no mercado de trabalho e contribuir significativamente para o avanço tecnológico.

VAMOS PRATICAR

1. No contexto da organização e armazenamento eficiente em bancos de dados e sistemas de arquivos, a árvore B desempenha um papel crucial. Essa estrutura é amplamente adotada devido à sua capacidade de proporcionar rápida inserção, remoção e busca, especialmente, em grandes conjuntos de dados. Assim, qual é a principal finalidade da árvore B em bancos de dados e sistemas de arquivos?

   a) Organizar dados de forma aleatória.
   b) Proporcionar rápida inserção, remoção e busca em grandes conjuntos de dados.
   c) Minimizar a eficiência de operações de busca.
   d) Aumentar o tempo de processamento em operações de inserção e remoção.
   e) Limitar o uso em sistemas de armazenamento de dados.

2. No contexto das estruturas de dados, a árvore B é amplamente utilizada para armazenar e organizar elementos de forma eficiente. Ela é composta por um conjunto de nós interligados, cada um contendo chaves ordenadas e ponteiros para outros nós na árvore. Considerando a estrutura básica de uma árvore B, quais são as características principais dos nós que a compõem? Analise as afirmativas a seguir:

   I - Chaves: são os valores armazenados na árvore e estão ordenados dentro de cada nó.
   II - Ponteiros: são referências para outros nós na árvore e permitem navegar de um nó para outro.
   III - Folhas: são os nós que não possuem filhos na árvore e armazenam apenas ponteiros.

   É correto o que se afirma em:

   a) I, apenas.
   b) III, apenas
   c) I e II, apenas.
   d) II e III, apenas.
   e) I, II e III.

3. A árvore B é uma estrutura de dados amplamente empregada em bancos e sistemas de arquivos devido à sua eficiência em organizar e manipular grandes conjuntos de elementos de forma rápida e eficaz. Dessa forma, qual das seguintes afirmações a respeito da árvore B é verdadeira?

   a) É uma estrutura de dados adequada apenas para pequenos conjuntos de dados.
   b) Foi projetada exclusivamente para operações de busca em sistemas de arquivos.
   c) Não permite operações de remoção de elementos.
   d) Reduz o número de comparações necessárias em operações de busca.
   e) É uma estrutura de dados estática, incapaz de lidar com alterações nos dados.

REFERÊNCIAS

BACKES, A. R. **Algoritmos e estruturas de dados em linguagem C**. Rio de Janeiro: LTC, 2023.

OLIVEIRA, P. M. de; PEREIRA, R. de L. **Estrutura de dados II**. Maringá: Unicesumar, 2019.

PEREIRA, S. do L. **Estruturas de dados em C**: uma abordagem didática. São Paulo: Érica, 2016.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 2010.

CONFIRA SUAS RESPOSTAS

1. Alternativa B.

   A principal finalidade da árvore B, em bancos de dados e sistemas de arquivos, é proporcionar rápida inserção, remoção e busca em grandes conjuntos de dados.

2. Alternativa C.

   Estão corretas apenas as afirmativas I e II.

3. Alterativa D.

   A árvore B reduz o número de comparações necessárias em operações de busca.

TEMA DE APRENDIZAGEM 7

ALGORITMOS DE ORDENAÇÃO: BUBBLE SORT, SELECTION SORT E INSERTION SORT

MINHAS METAS

- Compreender o funcionamento dos algoritmos de ordenação Bubble Sort, Selection Sort e Insertion Sort.
- Analisar a eficiência de cada algoritmo por meio da notação O.
- Implementar as sequências em uma linguagem de programação, levando em consideração a estrutura lógica de cada algoritmo de ordenação.
- Preparar-se para a concepção dos algoritmos de estruturas de dados em linguagem C.
- Avaliar o impacto dos algoritmos em ordenação em listas.
- Identificar as vantagens de cada algoritmo de ordenação.
- Desenvolver habilidades para solucionar problemas de ordenação em projetos de software a partir da construção em linguagem C dos algoritmos.

INICIE SUA JORNADA

Imagine que a agenda de seu celular não apresenta a função de busca por contatos e, para piorar, seus dados estão completamente desordenados. Não seria muito mais fácil organizar seus contatos em ordem alfabética para que nós, humanos, possamos utilizar a agenda de maneira mais adequada?

Suponha, agora, que você tem a lista de várias compras e vendas realizadas diariamente em uma organização. Ela se encontra ordenada de acordo com as datas nas quais cada compra ou venda ocorreu. Seu líder lhe solicita que organize a lista de forma a considerar o nome do comprador/vendedor em ordem alfabética. Como faria para ordenar sua lista?

A ordenação é um dos problemas mais fundamentais e frequentemente encontrados em ciência da computação. Este tema de aprendizagem aborda três dos algoritmos de ordenação mais básicos e importantes: Bubble Sort, Selection Sort e Insertion Sort. Eles são frequentemente ensinados nos cursos introdutórios de algoritmos e estruturas de dados devido à sua simplicidade e importância histórica. Apesar de não serem os mais eficientes para grandes volumes de dados, eles formam a base para a compreensão de algoritmos mais complexos.

A ordenação é essencial para várias operações em ciência da computação, como a busca eficiente e a análise de dados. Entender como os diferentes algoritmos funcionam pode ajudar a escolher o método mais apropriado para um determinado problema, melhorando, assim, a performance do sistema. Além disso, muitos algoritmos avançados de ordenamento são baseados em variações e otimizações desses métodos básicos.

Neste tema, exploramos os princípios de funcionamento do Bubble Sort, Selection Sort e Insertion Sort, discutindo suas complexidades e eficiência. Também, veremos como implementá-los e os utilizaremos em diferentes cenários práticos.

PLAY NO CONHECIMENTO

Neste podcast, falamos da importância dos algoritmos de ordenação, como eles funcionam e onde são aplicados. Também, abordamos sua evolução ao longo do tempo e a relevância do Bubble Sort, do Selection Sort e do Insertion Sort. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

VAMOS RECORDAR?

Que tal relembrarmos as árvores AVL? Elas são fundamentais para o completo entendimento de algoritmos de ordenação. Neste vídeo, você identificará como funciona esse elemento.

Acesse: **https://www.youtube.com/watch?v=YkF76cOgtMQ**.

DESENVOLVA SEU POTENCIAL

BUBBLE SORT

O **Bubble Sort**, ou **ordenação por bolha**, é um algoritmo simples de ordenação. Ele funciona repetidamente, passando pela lista a ser ordenada, comparando elementos adjacentes e trocando-os se estiverem na ordem errada. Esse processo é refeito até que a lista esteja ordenada. Embora seja intuitivo e fácil de imple-

mentar, o Bubble Sort não é eficiente para grandes conjuntos de dados devido à sua complexidade de tempo de O(n^2). Essa notação de eficiência é chamada de Big O. Ela é usada para descrever como o tempo de execução de um algoritmo cresce à medida que o tamanho da entrada (n) aumenta (Oliveira; Pereira, 2019).

Apesar de suas limitações, o Bubble Sort é útil para **fins educacionais**, pois ajuda os estudantes a entenderem os conceitos básicos de algoritmos de ordenação. Além disso, pode ser eficaz em pequenas listas ou listagens que já estão quase ordenadas, cujo desempenho pode ser comparável a algoritmos mais complexos.

INDICAÇÃO DE FILME

Matrix.

Matrix é um filme de ficção científica, que explora a ideia de uma realidade simulada controlada por máquinas inteligentes. A trama segue o protagonista, Neo, enquanto ele descobre que o mundo que conhece é uma simulação criada para manter a humanidade subjugada.

Refletindo sobre a história: a simulação da Matrix pode ser comparada a uma estrutura de árvore de decisão, em que cada escolha ou ação de um indivíduo leva a diferentes caminhos e resultados, muito parecido com o funcionamento de uma árvore de decisão em algoritmos de *machine learning*. Os agentes dentro da Matrix e os vários programas que controlam o mundo simulado, por sua vez, podem ser vistos como nós em um grafo, com conexões que representam as interações entre eles. Isso remete a estruturas de grafos usadas em algoritmos de busca e redes.

Na prática, o Bubble Sort é raramente usado em aplicações do mundo real . Entretanto, compreender seu funcionamento e limitações é crucial para desenvolver uma base sólida em algoritmos e estrutura de dados (Quadro 1) (Backes, 2023).

O Bubble Sort é raramente usado em aplicações do mundo real

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // troca arr[j] e arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Quadro 1 – Bubble Sort em C
Fonte: os autores.

No Quadro 1, contendo o código acima, o algoritmo itera sobre a lista, comparando cada par de elementos adjacentes. Se estiverem na ordem errada, eles são trocados. A variável *swapped* é usada para otimizar o algoritmo, parando a execução se a lista já estiver ordenada.

Suponha que temos uma lista de números [5, 3, 8, 4, 2]. Usando o Bubble Sort, começamos, comparando os dois primeiros elementos. Como 5 é maior que 3, os trocamos, resultando na lista [3, 5, 8, 4, 2]. Continuamos comparando e trocando elementos adjacentes. No final da primeira iteração, o maior elemento, 8, estará no final da lista. Repetimos o processo, ignorando o último número a cada nova iteração, até que a lista esteja completamente ordenada.

Após a segunda iteração, a lista se transforma em [3, 5, 4, 2, 8]. Após a terceira, a lista é [3, 4, 2, 5, 8]. Continuando, temos [3, 2, 4, 5, 8] e, finalmente, [2, 3, 4, 5, 8]. Cada elemento é comparado e trocado se necessário, até que não sejam necessárias mais trocas, indicando que a lista está ordenada.

Embora algoritmos, como Bubble Sort, Selection Sort e Insertion Sort, sejam úteis para entender os conceitos básicos de ordenação, eles não são os mais eficientes para grandes volumes de dados. Algumas otimizações podem ser aplicadas para melhorar seu desempenho (Pereira, 2016).

ZOOM NO CONHECIMENTO

No Bubble Sort, por exemplo, pode-se adicionar uma verificação para parar o algoritmo se a lista já estiver ordenada em uma iteração. No Insertion Sort, o uso da pesquisa binária para encontrar a posição correta para inserção pode reduzir o número de comparações necessárias.

SELECTION SORT

O **Selection Sort**, ou **ordenação por seleção**, é um algoritmo que divide a lista em duas sublistas: a dos itens já ordenados e a dos elementos restantes a serem ordenados.

ZOOM NO CONHECIMENTO

A **sublista dos itens já ordenados** é construída da esquerda para a direita no início da lista, e a **sublista dos itens restantes a serem ordenados** ocupam o resto da listagem. Inicialmente, a sublista ordenada está vazia e a sublista não ordenada é a lista completa.

O algoritmo funciona, repetidamente, selecionando o menor (ou maior, dependendo da ordem desejada) elemento da sublista não ordenada, trocando-o com o primeiro elemento não ordenado, e movendo os limites da sublista ordenada um elemento à frente.

A principal vantagem do Selection Sort é sua simplicidade e facilidade de implementação. Além disso, ele realiza o mínimo possível de trocas, o que pode ser uma vantagem em casos em que a operação de troca é cara. Entretanto, sua complexidade de tempo de O(n^2) o torna inadequado para grandes listas.

Embora não seja o algoritmo mais eficiente, o Selection Sort tem suas aplicações (Szwarcfiter; Markenzon, 2010).

Pode ser útil em cenários em que o custo de comparações é baixo em relação ao valor de trocas, ou quando a simplicidade do código é um fator importante (Backes, 2023).

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_idx = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // troca arr[min_idx] e arr[i]
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Quadro 2 – Selection Sort em C
Fonte: os autores.

No Quadro 2, contendo o código acima, o algoritmo procura o menor elemento na sublista não ordenada e o troca com o primeiro elemento não ordenado. Esse processo é repetido até que a lista esteja ordenada.

Vamos ordenar a lista [29, 10, 14, 37, 13], usando Selection Sort. Em primeiro lugar, encontramos o menor elemento na lista, 10, e o trocamos com o primeiro elemento, 29. A lista agora é [10, 29, 14, 37, 13]. Em seguida, procuramos o menor elemento na sublista restante [29, 14, 37, 13], que é 13, e o trocamos com o segundo elemento, 29, resultando na lista [10, 13, 14, 37, 29].

Continuamos o processo para os elementos restantes. O menor item na sublista [14, 37, 29] é 14, que já está na posição correta. Em seguida, o menor elemento na sublista [37, 29] é 29, que é trocado com 37. A lista final ordenada é [10, 13, 14, 29, 37]. Cada etapa envolve encontrar o menor item e colocá-lo na posição correta (Oliveira; Pereira, 2029).

APROFUNDANDO

A estabilidade de um algoritmo de ordenação é uma característica importante a ser considerada, pois é responsável por manter a ordem relativa dos registros com chaves iguais. Isso é essencial em situações em que essa sequência tem significado, como listas de objetos complexos ordenados por múltiplos critérios. O Insertion Sort é um exemplo de algoritmo estável, enquanto o Selection Sort não é. Entender quando a estabilidade é necessária ajuda na escolha da opção adequada para cada aplicação.

INSERTION SORT

O **Insertion Sort**, ou **ordenação por inserção**, é um algoritmo que constrói a lista ordenada de um item de cada vez. Ele é muito eficiente para pequenas quantidades de dados e pode ser mais eficiente que o Bubble Sort e o Selection Sort em muitos casos práticos.

O Insertion Sort é semelhante a como os humanos normalmente ordenam uma mão de cartas. Começamos com uma mão vazia, pegamos as cartas da mesa uma a uma e as inserimos na posição correta na mão.

O Insertion Sort tem uma complexidade de tempo de O(n^2) no pior caso, mas pode ter um desempenho muito melhor em listas que já estão quase ordenadas. Em tais casos, ele pode funcionar em tempo linear, O(n). Isso faz do Insertion Sort uma escolha excelente para listas pequenas ou para listas que são quase ordenadas (Pereira, 2019).

Além disso, o Insertion Sort é um **algoritmo estável**, o que significa que ele preserva a ordem relativa dos registros com chaves iguais. Essa propriedade é importante em muitas aplicações em que a ordem dos dados importa. Embora não seja adequado para grandes conjuntos de dados, o Insertion Sort é uma ferramenta valiosa em muitas situações práticas (Backes, 2023).

O Insertion Sort é uma ferramenta valiosa em muitas situações práticas

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        // Move os elementos de arr[0..i-1], que são
maiores que a chave, para uma posição à frente de sua
posição atual
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Quadro 2 - Selection Sort em C
Fonte: os autores.

No código acima, o algoritmo seleciona um elemento da lista e o insere na posição correta na sublista ordenada. Esse processo é repetido para cada elemento até que a listagem esteja ordenada.

Para a lista [9, 5, 1, 4, 3], o Insertion Sort começa com o segundo elemento. Comparamos 5 com 9, e como 5 é menor, movemos 9 para a direita e inserimos 5 na primeira posição, resultando em [5, 9, 1, 4, 3]. Em seguida, 1 é comparado com 9 e 5. Ambos são maiores, então, 1 é movido para a primeira posição, resultando em [1, 5, 9, 4, 3].

O próximo elemento 4 é comparado com 9 e 5, ambos são maiores, então, 4 é inserido antes deles, resultando em [1, 4, 5, 9, 3]. Finalmente, 3 é comparado com 9, 5, e 4, sendo posicionado antes deles, resultando na lista ordenada [1, 3, 4, 5, 9]. Cada elemento é colocado na posição correta em relação aos itens já ordenados (Oliveira; Pereira, 2019).

Em ciência de dados, a ordenação das informações é uma etapa crucial no pré-processamento. Algoritmos de ordenação são usados para limpar e organizar elementos antes de realizar análises mais profundas ou treinar modelos de aprendizado de máquina.

Ao preparar dados para um algoritmo de regressão, por exemplo, pode ser necessário ordenar os elementos por uma variável específica. Além disso, a visualização de informações, como gráficos de linha, frequentemente requer que os dados estejam ordenados para serem interpretados corretamente.

EM FOCO

Confira a aula referente a este tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

NOVOS DESAFIOS

Os algoritmos de ordenação são fundamentais para muitas aplicações no mundo real. Em bancos de dados, por exemplo, a ordenação é usada para organizar registros para busca eficiente. Em ciência de dados, algoritmos de ordenação são usados para preparar informações para visualizações e análises. No desenvolvimento de software, compreender como e quando usar diferentes algoritmos de ordenação pode melhorar significativamente a eficiência e a performance de suas aplicações.

Em sistemas de banco de dados, a ordenação de registros permite que as operações de busca e consulta sejam realizadas de maneira mais rápida e eficiente. Por exemplo, a execução de uma consulta que busca por um intervalo de valores pode ser acelerada significativamente se os dados estiverem previamente ordenados. Isso é essencial para aplicações que exigem tempos de resposta rápidos, como sistemas de transações financeiras e motores de busca na internet.

Dados ordenados, por sua vez, facilitam a identificação de padrões, *outliers* e tendências. Além disso, alguns algoritmos de aprendizado de máquina, como árvores de decisão, podem se beneficiar de elementos que são ordenados de maneira apropriada, melhorando a precisão e a eficiência do modelo.

No desenvolvimento de software, escolher o **algoritmo de ordenação** adequado pode ter um impacto significativo no desempenho de um programa. Em situações em que a memória é limitada ou quando a velocidade é crítica, selecionar o algoritmo certo pode fazer a diferença entre uma aplicação eficiente e uma que é lenta e consome muitos recursos. Além disso, entender as características dos diferentes algoritmos de ordenação ajuda os desenvolvedores a fazer *trade-offs* informados entre tempo de execução e uso de memória.

Outro aspecto importante é a ordenação em tempo real, que é essencial em sistemas embarcados e aplicações de streaming de dados. Nesses contextos, os algoritmos de ordenação devem ser capazes de lidar com fluxos contínuos de dados e realizar a ordenação de maneira incremental. Isso exige algoritmos que não apenas sejam eficientes, mas que também possam ser facilmente adaptados para trabalhar em um ambiente de dados em tempo real.

VAMOS PRATICAR

1. Em uma aplicação simples, que processa listas pequenas de dados, o desenvolvedor precisa escolher um algoritmo de ordenação adequado que seja fácil de implementar e eficiente para listas pequenas. Sendo assim, qual dos seguintes algoritmos de ordenação é geralmente mais eficiente para pequenas listas que já estão quase ordenadas?

   a) Bubble Sort.
   b) Selection Sort.
   c) Insertion Sort.
   d) Merge Sort.
   e) Heap Sort.

2. Durante uma aula de ciência da computação, os alunos foram apresentados a vários algoritmos de ordenação e suas características. O professor explicou que alguns algoritmos são mais eficientes que outros em diferentes situações e pediu que os alunos identificassem as características específicas dos algoritmos estudados. Sendo assim, qual dos seguintes algoritmos de ordenação realiza o mínimo possível de trocas, mas ainda tem complexidade de tempo O(n^2)?

   a) Bubble Sort.
   b) Selection Sort.
   c) Insertion Sort.
   d) Quick Sort.
   e) Merge Sort.

3. Em um projeto acadêmico, um estudante precisa ordenar uma pequena lista de números de forma eficiente e com o mínimo de complexidade. Ele está considerando usar um algoritmo que seja simples de entender e implementar. Dessa forma, qual dos seguintes algoritmos de ordenação é o mais simples de implementar, mas pode ser ineficiente para listas grandes?

   a) Bubble Sort.
   b) Selection Sort.
   c) Insertion Sort.
   d) Heap Sort.
   e) Merge Sort.

REFERÊNCIAS

BACKES, A. R. **Algoritmos e estruturas de dados em linguagem C**. Rio de Janeiro: LTC, 2023.

OLIVEIRA, P. M. de; PEREIRA, R. de L. **Estrutura de dados II**. Maringá: Unicesumar, 2019.

PEREIRA, S. do L. **Estruturas de dados em C**: uma abordagem didática. São Paulo: Érica, 2016.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 2010.

CONFIRA SUAS RESPOSTAS

1. Alternativa C.

   A alternativa A está incorreta, pois, embora o Bubble Sort seja fácil de implementar, ele é geralmente ineficiente mesmo para pequenas listas, uma vez que seu tempo de execução, no pior caso, é O(n^2), e ele requer muitas trocas, tornando-o ineficiente em listas quase ordenadas.
   A alternativa B está incorreta, pois o Selection Sort sempre percorre a lista inteira para encontrar o menor elemento, o que também resulta em um tempo de execução de O(n^2), e ele é menos eficiente para listas quase ordenadas.
   A alternativa C está correta, pois o Insertion Sort é muito eficiente para listas pequenas e quase ordenadas, uma vez que ele só precisa fazer algumas comparações e poucas trocas. Sua complexidade de tempo, no melhor caso, é O(n) quando a lista já está quase ordenada.
   A alternativa D está incorreta, pois o Merge Sort tem uma complexidade de tempo de O(nlog n), mas ele é mais complexo de implementar e não é especialmente eficiente para listas pequenas ou quase ordenadas devido ao overhead das operações de mesclagem.
   A alternativa E está incorreta, pois o Heap Sort também tem uma complexidade de O(nlog n) e é mais adequado para listas maiores. Para listagens pequenas, o overhead de construir o heap faz com que ele seja menos eficiente do que o Insertion Sort.

2. Alternativa B.

   A alternativa A está incorreta, pois o Bubble Sort realiza muitas trocas, uma vez que ele constantemente compara e troca elementos adjacentes. Assim, ele não minimiza o número de trocas.
   A alternativa B está correta, pois o Selection Sort faz o mínimo número de trocas possível entre todos os algoritmos de tempo O(n^2). Ele seleciona o menor elemento e faz apenas uma troca por iteração.
   A alternativa C está incorreta, pois o Insertion Sort faz mais trocas do que o Selection Sort, uma vez que ele constantemente move os elementos enquanto ordena a lista, embora seja eficiente para listas quase ordenadas.
   A alternativa D está incorreta, pois, embora o Quick Sort seja mais rápido em média com complexidade O(nlog n), ele não se qualifica como um algoritmo de tempo O(n^2) e pode realizar muitas trocas, especialmente, no pior caso.
   A alternativa E está incorreta, pois o Merge Sort não realiza trocas no sentido tradicional, mas sim divide e mescla listas. Ele tem uma complexidade O(nlog n) não O(n^2).

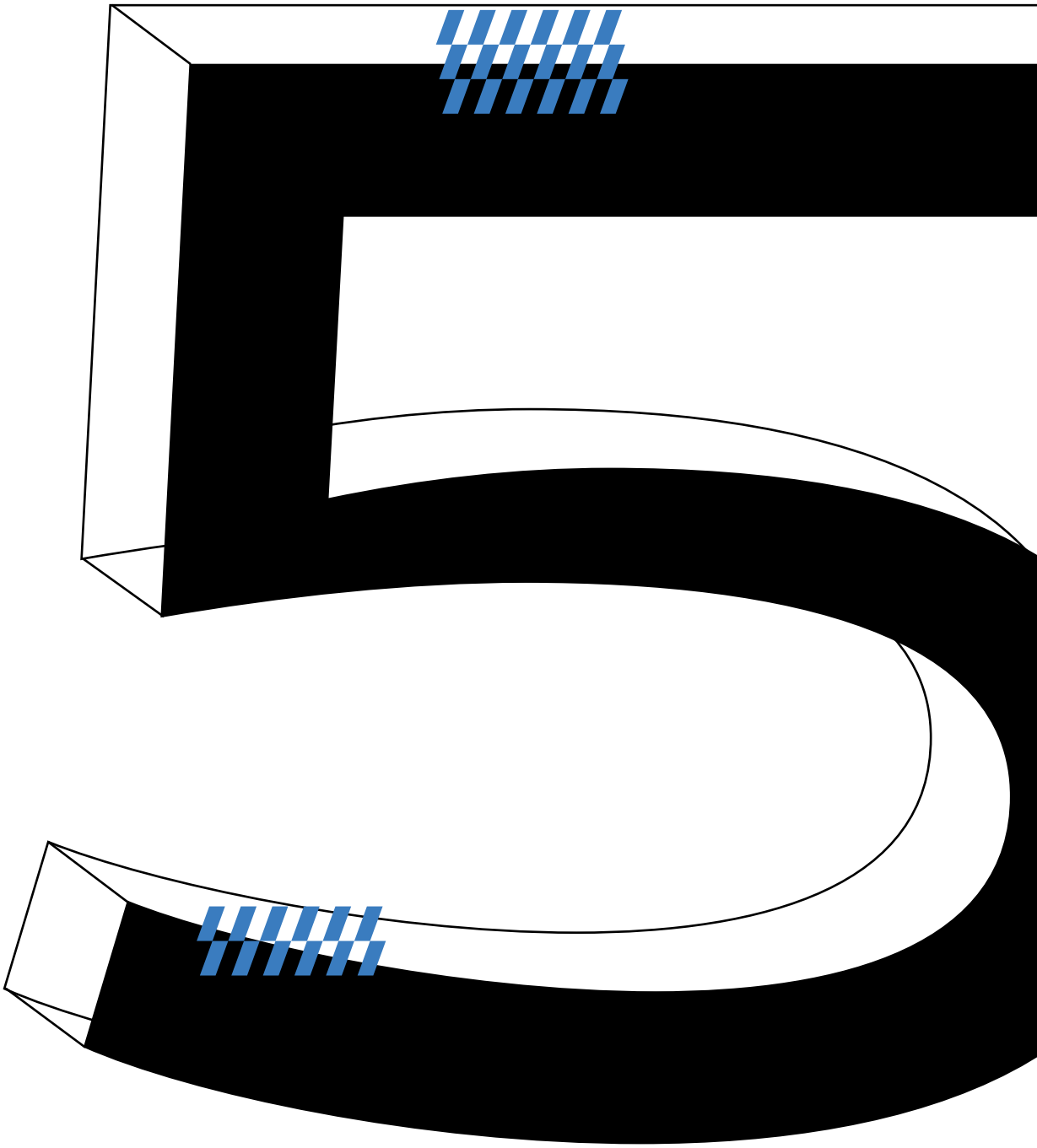
3. Alternativa A.

A alternativa A está correta, pois o Bubble Sort é extremamente simples de implementar, uma vez que ele apenas compara e troca elementos adjacentes repetidamente. Entretanto, ele é ineficiente para listas grandes devido ao seu tempo de execução O(n^2).
A alternativa B está incorreta, pois, embora simples, o Selection Sort é um pouco mais complexo que o Bubble Sort. Ele também é ineficiente para listas grandes com tempo O(n^2).
A alternativa C está incorreta, pois o Insertion Sort é relativamente simples, e é mais eficiente que o Bubble Sort para listas pequenas ou quase ordenadas. Entretanto, ele ainda tem complexidade O(n^2) no pior caso.
A alternativa D está incorreta, pois o Heap Sort é mais eficiente para listas grandes com O(nlog n), mas é mais complexo de implementar devido à necessidade de construir e manter um heap.
A alternativa E está incorreta, pois o Merge Sort é eficiente com O(nlog n) e adequado para listas grandes, mas é mais complexo de implementar devido à necessidade de divisão e mesclagem recursiva das sublistas.

MEU ESPAÇO

Unicesumar

unidade



TEMA DE APRENDIZAGEM 8

ALGORITMOS DE ORDENAÇÃO AVANÇADOS

MINHAS METAS

- Compreender o funcionamento dos algoritmos Shell Sort.
- Entender o funcionamento do algoritmo Merge Sort.
- Analisar o funcionamento do algoritmo Quick Sort.
- Verificar o funcionamento do algoritmo Heap Sort.
- Examinar a eficiência e complexidade de cada algoritmo.
- Implementar os algoritmos em uma linguagem de programação C.
- Identificar as vantagens de cada algoritmo de ordenação.

INICIE SUA JORNADA

Imagine que você é um desenvolvedor responsável pela criação de um sistema de gerenciamento de estoque para uma grande rede de lojas. Esse software deve ser capaz de organizar e processar milhares de produtos em questão de segundos para garantir que os dados de estoque estejam sempre atualizados em tempo real. Durante a fase de testes, você percebe que o programa está apresentando lentidão significativa ao tentar ordenar grandes volumes de elementos, o que pode causar atrasos críticos nas operações dos estabelecimentos. Como resolver essa questão de forma eficiente?

A importância de resolver essa problemática é fundamental. A eficiência no processamento de dados não só garante que o sistema de estoque funcione corretamente, mas também impacta diretamente na experiência dos clientes e na produtividade dos funcionários. Em um cenário em que o tempo é um recurso valioso, a capacidade de organizar dados rapidamente pode significar a diferença entre o sucesso e o fracasso de uma operação. Além disso, entender e aplicar algoritmos de ordenação avançados permite que os desenvolvedores otimizem seus sistemas, tornando-os mais robustos e escaláveis.

PLAY NO CONHECIMENTO

Neste podcast, discutimos a importância dos algoritmos Shell Sort, Merge Sort, Quick Sort e Heap Sort, suas aplicações no mundo real, assim como as opções indicadas aos diferentes cenários. Falamos acerca da eficiência de cada algoritmo e das técnicas para otimizá-los. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

VAMOS RECORDAR?

Antes de avançarmos, recordemo-nos de alguns conceitos fundamentais acerca dos algoritmos de ordenação. É importante compreender como operam tais sequências para entender o funcionamento dos algoritmos de ordenação avançados.

https://www.youtube.com/watch?v=t9_dHIy0QxA

DESENVOLVA SEU POTENCIAL

SHELL SORT

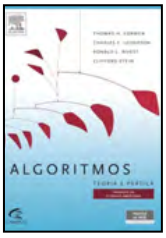
O **Shell Sort** é uma generalização do Insertion Sort que permite a troca de itens, os quais estão distantes uns dos outros. Assim, ao trocar elementos distantes, o algoritmo pode mover elementos para suas posições corretas mais rapidamente do que se movesse apenas itens adjacentes (Backes, 2023).

O Shell Sort melhora o desempenho do Insertion Sort ao comparar elementos que estão a um certo intervalo uns dos outros, em vez de confrontar itens adjacentes. A escolha do intervalo, ou gap, é crítica para a eficiência do Shell Sort. Inicialmente, um grande intervalo é utilizado, reduzindo-se gradualmente até que o gap seja 1, momento em que o algoritmo se transforma no Insertion Sort (Backes, 2023).

INDICAÇÃO DE LIVRO

Algoritmos

Esta obra é a tradução de *Introduction to Algorithms*, uma referência clássica, que cobre de forma abrangente muitos tipos de algoritmos, incluindo os de ordenação.



A complexidade do Shell Sort depende da escolha do intervalo . Em muitos casos, o desempenho é substancialmente melhor que o do Insertion Sort. Entretanto, a escolha inadequada do gap pode resultar em performance pior.

A complexidade do Shell Sort depende da escolha do intervalo

Observemos o Quadro 1:

```
void shellSort(int arr[], int n) {
    for (int gap = n/2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i += 1) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
```

Quadro 1 – Insertion Sort em C
Fonte: os autores.

Considere a lista [23, 29, 15, 19, 31, 7, 9, 5, 2]. Utilizando um intervalo inicial de 4, o Shell Sort compara e troca elementos que estão a quatro posições de distância. Após essa fase, a lista pode parecer [19, 2, 15, 5, 23, 7, 9, 29, 31]. Em seguida, com um intervalo de dois, e finalmente um, a lista será gradualmente ordenada até se tornar [2, 5, 7, 9, 15, 19, 23, 29, 31].

MERGE SORT

O **Merge Sort** é um algoritmo de ordenação baseado na técnica de divisão e conquista. Ele divide repetidamente a lista em duas metades, até que cada sublista contenha apenas um elemento, e, então, combina as sublistas de uma maneira ordenada para produzir a listagem final ordenada (Oliveira; Pereira, 2019).

O Merge Sort é particularmente eficiente para ordenar listas grandes, pois sua complexidade de tempo é O(n log n) no pior, no médio e no melhor caso. Isso faz do Merge Sort uma escolha popular quando se trata de ordenar grandes volumes de dados (Oliveira; Pereira, 2019).

Uma das características importantes do Merge Sort é que ele é um **algoritmo estável**, ou seja, mantém a ordem relativa dos registros com chaves iguais. Isso é especialmente útil em aplicações em que essa ordem importa.

Observe o Quadro 2:

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Quadro 1 – Insertion Sort em C
Fonte: os autores.

Considere a lista [38, 27, 43, 3, 9, 82, 10]. O Merge Sort divide a lista ao meio, resultando em [38, 27, 43] e [3, 9, 82, 10]. Essas sublistas são divididas novamente até que cada um delas contenha um único elemento. Elas são, então, combinadas de forma ordenada, resultando na listagem final [3, 9, 10, 27, 38, 43, 82].

O Merge Sort utiliza a técnica de divisão e conquista para ordenar uma lista. O algoritmo divide a lista repetidamente até que cada sublista contenha apenas um elemento, e, então, combine as sublistas de forma ordenada para produzir a listagem final estruturada. Essa abordagem garante uma complexidade de tempo de O(n log n), tornando o Merge Sort eficiente para grandes volumes de dados. Além disso, ele é um algoritmo estável, que preserva a ordem relativa dos registros com chaves iguais, o que é vantajoso em várias aplicações. Entretanto, requer espaço adicional para as sublistas, o que pode ser uma desvantagem em sistemas com memória limitada (Pereira, 2016).

QUICK SORT

O **Quick Sort** é outro algoritmo de ordenação baseado na técnica de divisão e conquista. Ele seleciona um elemento como pivô e participa da lista em dois subarranjos: um com elementos menores que o pivô e outro com elementos maiores. Esse processo é repetido recursivamente para os subarranjos (Szwarcfiter, 2010).

O Quick Sort é conhecido por sua eficiência e é amplamente utilizado em aplicações em que a velocidade é crucial. Entretanto, sua complexidade de tempo, no pior caso, é O(n^2), o que pode ser problemático para listas já ordenadas ou com muitos elementos repetidos.

Para mitigar esse problema, estratégias, como a escolha de um pivô aleatório ou o uso da mediana de três, são frequentemente empregadas.

Uma característica importante do Quick Sort é que ele é um algoritmo instável, ou seja, não preserva a ordem relativa dos registros com chaves iguais. Entretanto, sua eficiência, em muitos casos, compensa essa desvantagem.

Observe o Quadro 3:

```
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
int partition (int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Quadro 3 – Quick Sort em C
Fonte: os autores.

A escolha do pivô no Quick Sort é um fator determinante para sua eficiência. Estratégias comuns incluem escolher o primeiro, o último, o do meio ou um elemento aleatório. Outra abordagem é a mediana de três, em que o pivô é escolhido como a mediana do primeiro, do meio e do último elemento. A escolha do pivô influencia diretamente o número de comparações e trocas necessárias. Se mal escolhido, ele pode resultar em uma divisão desequilibrada, levando a uma complexidade de tempo de $O(n^2)$. Entretanto, técnicas avançadas de escolha de pivô ajudam a manter a complexidade esperada de $O(n \log n)$.

Considere a lista [10, 7, 8, 9, 1, 5]. O Quick Sort escolhe 5 como pivô e a particiona em [1, 5, 8, 9, 7, 10]. A função é, então, chamada recursivamente para os subarranjos [1] e [8, 9, 7, 10]. Esse processo é repetido até que a listagem esteja completamente ordenada, resultando em [1, 5, 7, 8, 9, 10].

HEAP SORT

O **Heap Sort** é um algoritmo de ordenação que utiliza a estrutura de dados heap. Ele converte a lista em uma *heap* máxima, na qual o maior elemento é a raiz. O maior elemento é removido e colocado no final da lista, e a *heap* é ajustada para manter a sua propriedade. Esse processo é repetido até que todos os elementos estejam ordenados (Backes, 2023).

> Uma das principais vantagens do Heap Sort é que ele não requer espaço adicional além do necessário para a lista, ao contrário do Merge Sort, que requer espaço adicional para as sublistas. O Heap Sort é eficiente e tem uma complexidade de tempo de O(n log n), tanto no pior quanto no melhor caso (Backes, 2023).

Para implementar o Heap Sort, primeiramente, construímos uma heap máxima a partir dos dados a serem ordenados. Em seguida, repetidamente, removemos o maior elemento da *heap* (a raiz), o colocamos no final da lista ordenada e ajustamos a *heap* para manter sua propriedade. Esse processo continua até que a *heap* esteja vazia e a lista esteja ordenada.

Observe o Quadro 4:

```
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

void heapify(int arr[], int n, int i) {
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;
    if (l < n && arr[l] > arr[largest])
        largest = l;
    if (r < n && arr[r] > arr[largest])
        largest = r;
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n-1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
```

Quadro 4 – Heap Sort em C
Fonte: os autores.

Considere a lista [12, 11, 13, 5, 6, 7]. O Heap Sort, primeiramente, converte a lista em uma heap máxima, resultando em [13, 11, 12, 5, 6, 7]. O maior elemento 13 é removido e colocado no final da listagem. A heap é ajustada e o processo é repetido para os elementos restantes até que a lista esteja ordenada, resultando em [5, 6, 7, 11, 12, 13].

APROFUNDANDO

A **Heapificação** é o processo de transformar uma lista em uma *heap*, uma estrutura de dados que satisfaz a propriedade dela. Em uma *heap* máxima, para cada nó i que não seja uma folha, o valor do nó é maior ou igual aos valores de seus filhos. Esse processo é crucial para o funcionamento do Heap Sort, pois garante que o maior elemento da lista esteja sempre na raiz da *heap*.
A função *heapify* é chamada recursivamente para ajustar a estrutura da *heap* após a remoção do maior elemento, garantindo que a propriedade dela seja mantida. Esse método é eficiente, pois realiza operações de comparação e troca de maneira sistemática, permitindo uma ordenação eficaz.

EM FOCO

Confira a aula referente a este tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

NOVOS DESAFIOS

Para abordar o desafio apresentado anteriormente, podemos explorar diferentes algoritmos de ordenação avançados, como Quick Sort, Merge Sort e Heap Sort. Em um laboratório de programação, pode-se implementar esses algoritmos e testar suas eficácias em conjuntos de dados de diferentes tamanhos e complexidades. Por exemplo, envolver a ordenação de uma lista de produtos, baseada em critérios variados, como preço, data de validade ou popularidade. Ao comparar o desempenho de cada algoritmo, poderemos identificar qual abordagem oferece a melhor performance para um problema específico.

Nesse sentido, no mundo real, a habilidade de escolher e implementar o algoritmo correto pode otimizar processos, economizar recursos e melhorar a experiência do usuário em diferentes contextos. Por exemplo, em empresas de tecnologia, a eficiência de um algoritmo de ordenação pode determinar a velocidade e a eficácia de serviços on-line, como motores de busca e plataformas de e-commerce.

Além do impacto imediato na performance dos sistemas, o conhecimento em algoritmos de ordenação é uma habilidade valiosa no mercado de trabalho. Profissionais com essa expertise são mais competitivos, pois conseguem desenvolver soluções mais robustas e escaláveis. A capacidade de resolver problemas complexos com eficiência é altamente valorizada em áreas, como ciência de dados, engenharia de software e desenvolvimento de sistemas.

Mais amplamente, o estudo e a aplicação de algoritmos de ordenação promovem uma compreensão mais profunda da ciência da computação. Há um pensar em forma lógica e estruturada, habilidades que são essenciais para qualquer carreira técnica. Eles também desenvolvem a capacidade de analisar e comparar diferentes abordagens para um problema, o que é uma competência crítica não apenas em programação, mas em qualquer campo que envolva tomada de decisão baseada em dados.

VAMOS PRATICAR

1. Em um grande banco de dados, a eficiência das operações de busca e consulta é fundamental para garantir um desempenho rápido. A escolha do algoritmo de ordenação correto pode melhorar significativamente o tempo de resposta do sistema. Considere que você precisa ordenar uma lista de transações financeiras para facilitar a busca por valores específicos. Qual dos seguintes algoritmos de ordenação é mais adequado para garantir um desempenho consistente e eficiente em grandes volumes de dados, sem a necessidade de espaço adicional significativo?

   a) Shell Sort.
   b) Merge Sort.
   c) Quick Sort.
   d) Heap Sort.
   e) Bubble Sort.

2. Durante uma aula de ciência da computação, os alunos estão aprendendo diferentes algoritmos de ordenação. O professor destacou que alguns deles são mais eficientes para listas grandes, enquanto outros são mais simples de implementar. Entretanto, a eficiênciaw e a simplicidade nem sempre estão alinhadas. Qual dos seguintes algoritmos de ordenação oferece uma complexidade de tempo de O(n log n) em todos os casos, mas não é um algoritmo estável?

   a) Shell Sort.
   b) Merge Sort.
   c) Quick Sort.
   d) Heap Sort.
   e) Insertion Sort.

3. Em uma aplicação de ciência de dados, é necessário preparar um grande conjunto de elementos para análise estatística. O pesquisador precisa garantir que as informações sejam ordenadas de forma eficiente para facilitar a identificação de padrões e tendências. Qual dos seguintes algoritmos de ordenação é uma escolha robusta para grandes conjuntos de dados devido à sua complexidade de tempo de O(n log n) e por ser um algoritmo estável?

   a) Shell Sort.
   b) Merge Sort.
   c) Quick Sort.
   d) Heap Sort.
   e) Bubble Sort.

REFERÊNCIAS

BACKES, A. R. **Algoritmos e estruturas de dados em linguagem C**. Rio de Janeiro: LTC, 2023.

OLIVEIRA; P. M. de; PEREIRA, R. de L. **Estrutura de dados II**. Maringá: Unicesumar, 2019.

PEREIRA, S. do L. **Estruturas de dados em C**: uma abordagem didática. São Paulo: Érica, 2016.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 2010.

CONFIRA SUAS RESPOSTAS

1. Alternativa D.

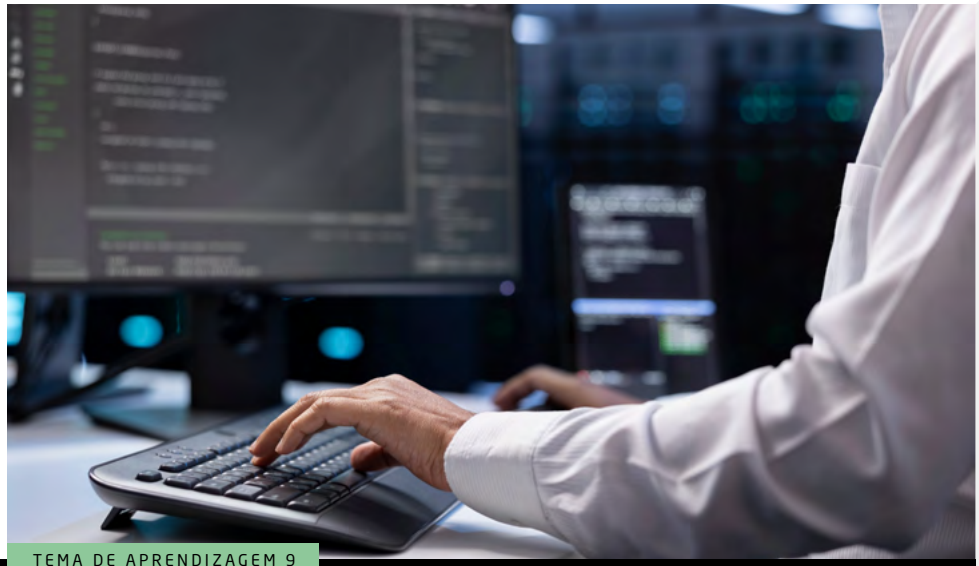
   O Heap Sort é um algoritmo de ordenação que oferece uma complexidade de tempo de O(n log n) em todos os casos e requer um espaço adicional mínimo, pois é um algoritmo *in-place*. Isso significa que ele ordena os dados sem a necessidade de utilizar espaço adicional significativo, o que o torna adequado para grandes volumes de dados. Além disso, o Heap Sort mantém um desempenho consistente, independentemente do estado inicial dos dados, ao contrário do Quick Sort, que pode ter um desempenho degradado no pior caso.

2. Alternativa D.

   O Heap Sort oferece uma complexidade de tempo de O(n log n) em todos os casos, o que garante um desempenho eficiente para listas grandes. Entretanto, ele não é um algoritmo estável, o que significa que não preserva a ordem relativa dos elementos iguais. Em comparação, Merge Sort e Quick Sort são algoritmos estáveis (embora Quick Sort possa não ser estável dependendo da implementação), e Shell Sort e Insertion Sort não garantem a complexidade de tempo O(n log n) em todos os casos.

3. Alternativa B.

   O Merge Sort é uma escolha robusta para grandes conjuntos de dados devido à sua complexidade de tempo de O(n log n) e por ser um algoritmo estável. Isso significa que ele preserva a ordem relativa dos elementos iguais, o que pode ser importante em análises estatísticas em que a integridade dos dados deve ser mantida. Embora Merge Sort possa requerer espaço adicional para a ordenação, sua estabilidade e desempenho consistente em todos os casos o tornam ideal para grandes volumes de dados.



TEMA DE APRENDIZAGEM 9

TABELA DE DISPERSÃO (*HASHING*)

MINHAS METAS

- Compreender o funcionamento de uma tabela de *hash*.
- Entender diferentes formas de resolução.
- Analisar a eficiência e complexidade das operações em tabelas de *hash*.
- Implementar uma tabela de *hash* em uma linguagem de programação.
- Aplicar tabelas de *hash* em diferentes cenários e conjuntos de dados.
- Avaliar o impacto do uso de tabelas de *hash* no desempenho de sistemas computacionais.
- Desenvolver habilidades para solucionar problemas de colisão em tabelas de *hash*.

INICIE SUA JORNADA

Imagine que você é responsável pelo desenvolvimento de um sistema de gerenciamento de usuários para uma grande rede social. O software precisa ser eficiente para armazenar, buscar e gerenciar milhões de usuários de forma rápida. Porém, à medida que a base de usuários cresce, as buscas se tornam mais lentas e a experiência dos usuários começa a se deteriorar. Como resolver esse problema de eficiência na busca de elementos em um banco de dados tão volumoso?

Resolver esse problema é fundamental para garantir a satisfação dos usuários e a competitividade da rede social. Se os tempos de resposta forem longos, as pessoas podem ficar frustradas e abandonar a plataforma, resultando em uma queda no engajamento e nos lucros. A implementação de uma tabela hash pode ser a chave para otimizar essas buscas, permitindo acessos quase instantâneos a dados específicos, independentemente do tamanho da base de usuários.

PLAY NO CONHECIMENTO

Neste podcast, discutimos a importância das tabelas de *hash*, como elas funcionam e quando são mais úteis. Falamos acerca de suas aplicações práticas, eficiência em comparação com outras estruturas de dados e dicas para implementá-las de forma eficiente. Abordamos, também, casos comuns de uso e resolução de problemas reais com o auxílio de tabelas de *hash*. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

VAMOS RECORDAR?

Antes de nos aprofundarmos no assunto deste tema, é importante relembrar alguns conceitos fundamentais acerca das estruturas de dados e das funções de hash. Revisitar a lógica de tais funções e as operações básicas em arrays nos ajudará a entender melhor as otimizações oferecidas pelas tabelas de hash. Assim, recordemo-nos de como funcionam as operações básicas de arrays e listas, acessando: **https://www.youtube.com/watch?v=jQ0r7P8rC1M**.

DESENVOLVA SEU POTENCIAL

ENTENDENDO TABELA DE DISPERSÃO

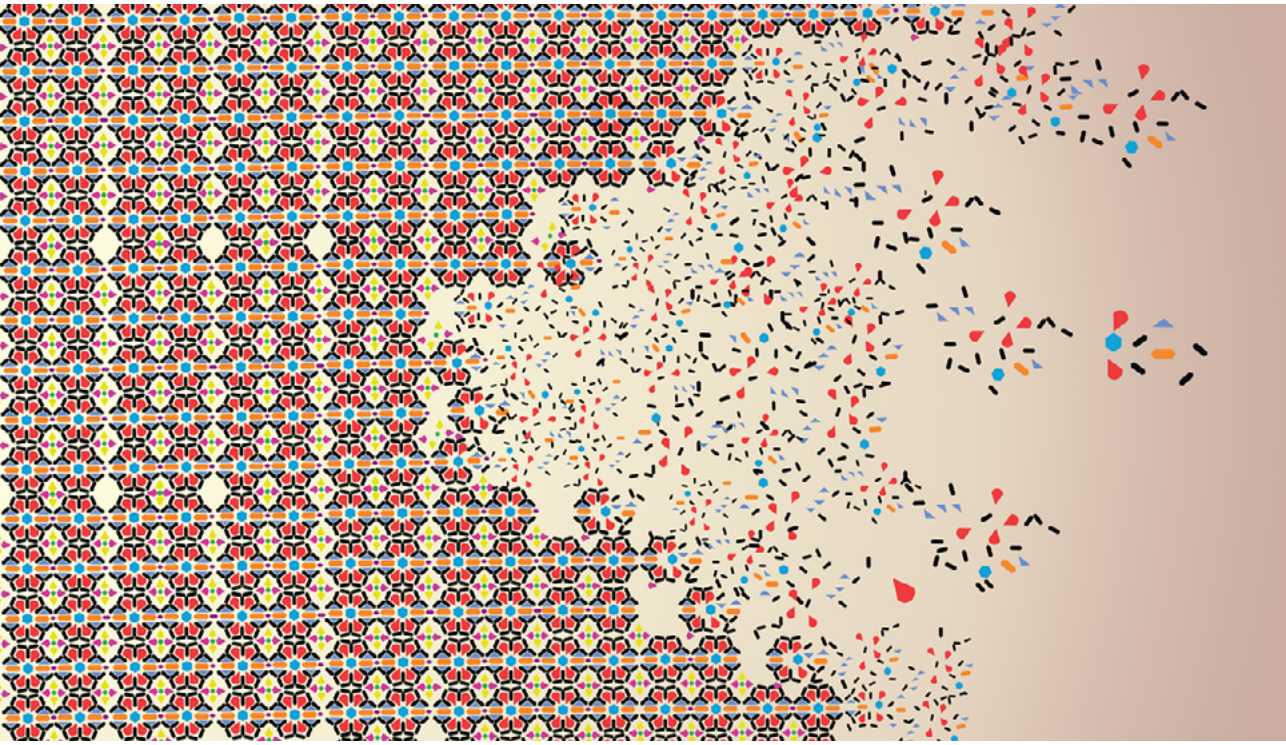
Drozdek (2008) comenta que é necessário desenvolver uma função h()h()h() que processe uma chave particular kkk em um índice que possa ser utilizado para armazenar itens em um vetor que seja do mesmo tipo de kkk (int, *float*, char etc.). Chama-se h()h()h() de função de dispersão, função de escrutínio ou função *hash*. Em suma, para nós, a função *hash* deve ser capaz de converter um elemento, ou um conjunto de dados, em um índice, que é a posição na qual tais informações serão armazenadas (Drozdek, 2008).

Em outras ocasiões, uma função *hash* pode gerar um código utilizado em aplicações que envolvam criptografia. Todavia, esse não é o foco de nosso estudo neste momento. A procura por interpolação pode ser mais rápida do que uma busca binária se os valores no vetor estiverem distribuídos de forma uniforme. Caso contrário, ela pode ser tão lenta como uma pesquisa sequencial (Backes, 2023; Szwarcfiter; Markenzon, 2010).

APROFUNDANDO

As funções de *hash* convertem uma chave em um índice no *array* em que os valores são armazenados. Uma boa função de *hash* deve ser rápida de calcular e distribuir as chaves uniformemente ao longo do *array* para minimizar colisões. Existem várias técnicas para criar funções de *hash* eficientes, incluindo métodos baseados em aritmética modular, multiplicação e divisão. A escolha da função pode impactar significativamente o desempenho de uma tabela de *hash*, especialmente, em aplicações com grandes volumes de dados.

Imagine, por exemplo, um vetor com dez posições, que vão de 0 a 9. Suponha que queiramos armazenar dados do tipo inteiro em nossa tabela de busca. Assim, poderíamos pensar em uma função h()h()h() tal que, realizando cálculos matemáticos sob o elemento a ser armazenado (ou buscado), cheguemos a um valor inteiro que varie entre 0 e 9, assim como o índice do vetor.



Considere, a título de ilustração, que nossa função h()h()h() recebe um parâmetro de entrada xxx, do tipo int, e retorna o cálculo da expressão "(x·x)%10(x \cdot x) \% 10(x·x)%10". Em outras palavras, nossa função *hash* retorna o resto de divisão do quadrado do parâmetro de entrada. Assim, para inserir o número 72 em nosso vetor, temos de realizar o cálculo "72·72%1072 \cdot 72 \% 1072·72%10", de acordo com a expressão de retorno de nossa função *hash*.

Dessa forma, ao realizar a conta, descobrimos que h(72)h(72)h(72) é igual a 4. Por isso, o elemento 72 deve ser armazenado na posição 4 do vetor. Matematicamente, podemos dizer que h(x)=x2mod 10h(x) = x^2 \mod 10h(x)=x2mod10, nesse caso.

Observe, na Figura 1, como ficou o vetor após a inserção dos elementos 11, 23, 26 e 72. O vetor que podemos visualizar na porção direita dessa imagem funciona como uma tabela, na qual os dados são armazenados de acordo com o resultado do processamento de uma chave pela função hash. Damos o nome a esse tipo de estrutura de dados de Tabela Hash, Tabela de Dispersão ou Tabela de Escrutínio.

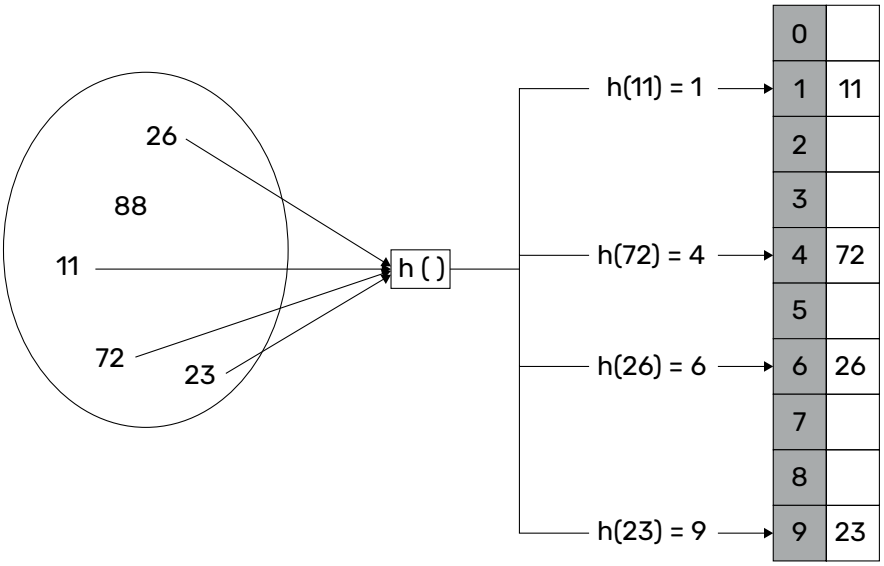


Figura 1 – Inserção de dados em uma Tabela *Hash*
Fonte: os autores.

Descrição da Imagem: a figura apresenta uma representação visual do processo de inserção de dados em uma tabela *hash*. Ela está dividida em duas partes principais: a função hash e a tabela *hash*. No lado esquerdo da imagem, há um círculo contendo vários números: 26, 88, 11, 72, 23, os quais são exemplos de dados que serão inseridos na tabela *hash*. Eles são conectados a um bloco central marcado como "h()", que representa a função *hash*. Essa função é responsável por calcular a posição de cada dado na tabela *hash*. À direita, há uma tabela vertical com índices de 0 a 9, que representa a tabela hash propriamente dita. Cada índice na tabela pode armazenar um valor. A imagem mostra como a função *hash* "h()" mapeia os números para os índices específicos da tabela. Cada linha que conecta a função *hash* "h()" a um índice na tabela indica como cada número será armazenado na tabela *hash*. Fim da descrição.

Para compreender melhor, tomemos o caso da inserção do elemento 23. Durante o processamento, a função h()h()h() encontra o quadrado de 23, que é igual a 529. Em seguida, a função *hash* divide 529 por 10, encontrando um quociente igual a 52 (que não nos interessa, nesse caso) e um resto de divisão igual a 9. Assim, o resultado da execução da função h()h()h(), passando o argumento 23, é 9. Por isso, o elemento 23 se encontra armazenado na posição 9 da tabela. Esse mesmo processo é repetido para todo elemento que é inserido.

Após inseridos os elementos em nosso vetor, para realizar uma busca, basta utilizar a função *hash* novamente, de maneira similar ao que foi feito no momento da inserção. Em outras palavras, se quisermos realizar uma busca pelo elemento 11, por exemplo, devemos invocar a função *hash* para que, nesse caso, saibamos que o elemento 11 se encontra na posição 1 da tabela.

Entretanto, se tentássemos buscar um elemento ausente, digamos, o 35, o que aconteceria? Bem, ao processar o elemento 35 de acordo com nossa função *hash*, descobriríamos que o 35 deveria estar armazenado na posição 5. Todavia, como vemos na Figura 2, o 35 não se encontra armazenado nessa posição. Assim, o algoritmo de busca conclui que o elemento está ausente.

Apesar disso tudo, será que não há algum problema com nossa tabela? Repare que, na Figura 2, o elemento 88 não foi armazenado no vetor, propositalmente. Ao tentar inserir o elemento 88, verificaríamos que o resultado de h(88)h(88)h(88) é igual a 4. Com isso, teríamos de sobrescrever o dado que se encontra armazenado na posição 4. Damos o nome de colisão a fatos como esse. Sempre que duas ou mais chaves, depois de processadas pela função *hash*, apresentam um resultado igual, temos uma colisão.

Existem diversas técnicas para resolver o problema das colisões em tabelas *hash*. Pense que, quando houver uma colisão, decidimos procurar pela próxima posição livre no vetor para armazenar o dado conflitante. Dessa forma, ao fazer a busca, ao calcular o *hash* da chave, não bastaria apenas verificar a posição indicada pela função *hash*, mas também as posições subsequentes (Pereira, 2016).

APROFUNDANDO

O encadeamento separado é uma técnica popular para tratar colisões em tabelas de *hash*. Em vez de encontrar outra posição no *array*, como no endereçamento aberto, o encadeamento separado usa listas ligadas para armazenar múltiplos valores no mesmo índice. Cada entrada na tabela de *hash* contém um ponteiro para uma lista ligada de elementos que compartilham o mesmo índice de *hash*. Essa abordagem facilita a inserção e remoção de elementos, mas pode levar a listas longas e desbalanceadas se a função de *hash* não distribuir as chaves uniformemente.

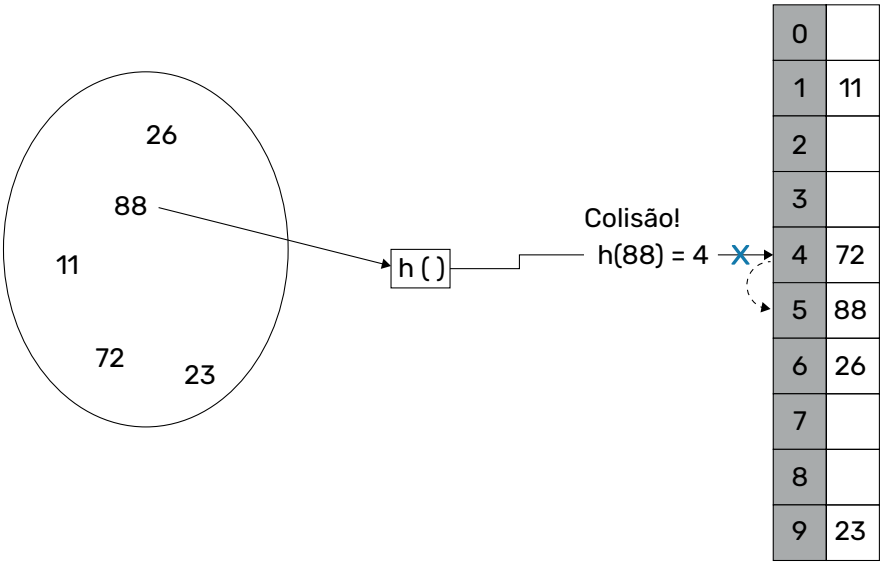
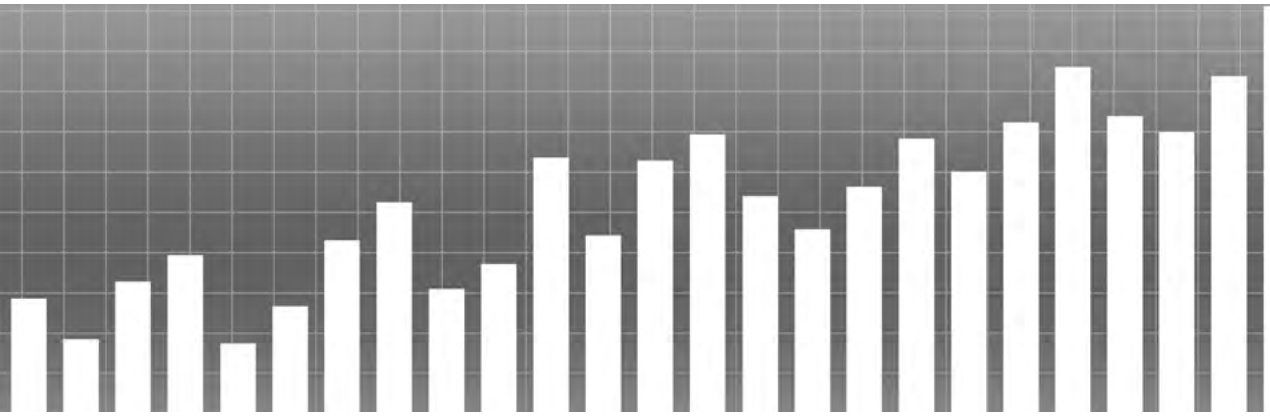


Figura 2 – Resolvendo o problema de colisão em uma inserção
Fonte: os autores.

Descrição da Imagem: a figura ilustra o problema de colisão e sua resolução durante a inserção de dados em uma tabela *hash*. Ela está dividida em duas partes principais: a função *hash* e a tabela *hash*. No lado esquerdo da imagem, há um círculo contendo vários números: 26, 88, 11, 72, 23, os quais representam os dados que serão inseridos na tabela *hash*. Eles são conectados a um bloco central marcado como "h()", que representa a função *hash*, a qual calcula a posição de cada dado na tabela *hash*. À direita, há uma tabela vertical com índices de 0 a 9, que representa a tabela *hash*. A tabela mostra como a função *hash* "h()" mapeia os números para os índices específicos da tabela. Na imagem, o número 88 é mapeado para o índice 4, o qual já contém o número 72, indicando uma colisão. Há uma seta indicando a colisão com a anotação "Colisão! h(88) = 4". A imagem sugere a resolução da colisão por meio do uso de encadeamento. Há uma seta pontilhada que mostra que o número 88 é inserido no índice 5, o próximo disponível. Isso demonstra uma técnica comum para resolver colisões em tabelas *hash*. Fim da descrição.

Essa é uma solução interessante, na medida em que a quantidade de dados a serem armazenados é menor que o tamanho do vetor estático. Quando tentamos inserir o 88 na posição 4 por meio da função *hash*, tivemos uma colisão. Nesse instante, nosso algoritmo pode tentar procurar pela próxima posição livre dentro do arranjo, após a colisão, para armazenar o elemento em questão.

Dessa forma, ao tentar realizar uma busca pelo elemento 88, a função *hash* retornaria à posição 4. Ao verificar a posição 4, notamos que quem se encontra lá é o 72. Todavia, nosso algoritmo sabe que pode ter havido alguma colisão. Por isso, a busca continua sequencialmente, verificando se existe algum elemento posterior ao 72 que seja igual ao 88. Para nossa sorte, o 88 está armazenado exatamente na posição seguinte, encerrando a busca com sucesso na posição 5. Entretanto, o algoritmo pode interromper a busca sequencial caso atinja alguma posição na qual não haja elemento armazenado.



Considere, ainda, que os elementos em uma tabela *hash* podem ser removidos, além de inseridos. Imagine que inserimos o 88 na posição 5 para solucionar a colisão, porém, em seguida, o elemento 72 é removido. Após isso, ao tentar buscar pelo elemento 88, tentaremos acessar a posição 4 devido à *função* hash e perceberemos que é uma posição vazia. A busca não pode ser interrompida quando um elemento está ausente por causa de uma remoção, pois o elemento-alvo pode ter sido inserido em um momento que ocasionava uma colisão. Por isso, nesse tipo de solução de colisões, é interessante que todo o vetor possua *flags* que sinalizem se uma posição vazia é resultado de uma remoção ou não (Oliveira; Pereira, 2019).

Outro método de solução de colisões consiste em combinar o caráter estático de vetores com o potencial de alocação dinâmico dos ponteiros. Assim, caso haja uma colisão, podemos "pendurar" mais elementos naquela posição do vetor de maneira dinâmica. Em outras palavras, o vetor passa a ser uma tabela na qual cada posição do vetor pode dar origem a uma lista encadeada distinta. Observe o problema da colisão do número 88 solucionada por meio de ponteiros, na Figura 3.

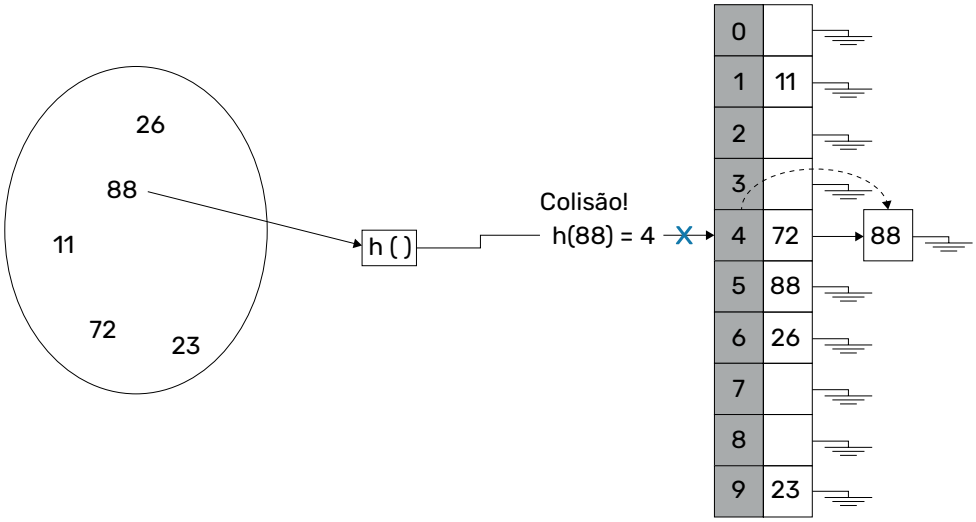


Figura 3 – Solucionando colisões por meio de alocação dinâmica
Fonte: os autores.

Descrição da Imagem: a figura ilustra a resolução de colisões em uma tabela *hash* por meio da alocação dinâmica, utilizando uma estrutura de encadeamento. A imagem é dividida em duas partes principais: a função *hash* e a tabela *hash*. No lado esquerdo, há um círculo contendo vários números: 26, 88, 11, 72, 23, os quais representam os dados que serão inseridos na tabela *hash*. Eles são conectados a um bloco central marcado como "h()", que representa a função *hash*, a qual calcula a posição de cada dado na tabela *hash*. À direita, há uma tabela vertical com índices de 0 a 9, que representa a tabela *hash*. A tabela mostra como a função *hash* "h()" mapeia os números para os índices específicos da tabela. A imagem foca no número 88, que é mapeado para o índice 4, o qual já contém o número 72, resultando em uma colisão. Há uma seta indicando a colisão com a anotação "Colisão! h(88) = 4". Para resolver a colisão, a imagem mostra o uso de uma estrutura de encadeamento. O número 88 é adicionado a uma lista vinculada ao índice 4. Há uma seta pontilhada que conecta o índice 4 ao número 88 em uma estrutura de lista encadeada. Isso significa que o índice 4, agora, aponta para uma lista que contém tanto o número 72 quanto o número 88. Fim da descrição.

A Figura 3 ilustra de maneira clara o que pode ser feito em caso de colisão. Antes de tudo, cada elemento deve ser um registro que contém ao menos um campo de "dados" e outro campo ponteiro para apontar para o próximo elemento (em caso de colisões). Assim, a Tabela *Hash* passa a ser um vetor de *structs* que pode ser flexível em seu tamanho, fazendo com que colisões resultem na criação de listas encadeadas.

APROFUNDANDO

As tabelas de *hash* são amplamente utilizadas em diversas áreas da ciência da computação e do desenvolvimento de software. Em bancos de dados, elas são usadas para indexar registros e acelerar buscas. Em sistemas de arquivos, tabelas de *hash* ajudam a gerenciar e localizar arquivos rapidamente. No desenvolvimento web, elas são essenciais para implementar caches, garantindo que dados frequentemente acessados possam ser recuperados rapidamente. Além disso, usadas em compilações de linguagem de programação para resolver colisões de nomes e gerenciar tabelas de símbolos.

EM FOCO

Confira a aula referente a este tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**.

NOVOS DESAFIOS

As **tabelas de *hash*** são uma **ferramenta poderosa** em **ciência da computação**, mas sua implementação e uso apresentam desafios únicos. À medida que os sistemas de software se tornam mais complexos e lidam com volumes crescentes de dados, compreender esses obstáculos e maneiras de superá-los se torna essencial para qualquer desenvolvedor.

Um dos principais **desafios** na implementação de tabelas de *hash* é a escolha da função de *hash*, que deve ser capaz de distribuir as chaves uniformemente ao longo do *array* de forma a minimizar colisões. Projetar uma função de *hash* que atenda a esses requisitos, no entanto, pode ser complicado, especialmente, quando se lida com diferentes tipos de dados e padrões de acesso. Funções de *hash* mal projetadas podem resultar em alta taxa de colisão, o que degrada o desempenho da tabela de *hash*.

Outro desafio significativo é a gestão de colisões. Embora existam várias técnicas para lidar com essas ocorrências, como endereçamento aberto e encadeamento separado, cada uma tem suas próprias vantagens e desvantagens.

O encadeamento separado, por exemplo pode simplificar a implementação e evitar problemas de *clustering* primário e secundário, entretanto, pode levar a listas longas e desbalanceadas que afetam o desempenho da busca. O endereçamento aberto, por sua vez, requer uma política de *probing* eficiente para evitar longas sequências de sondagens.

A **eficiência espacial** também é uma consideração importante. Tabelas de *hash* geralmente precisam ser dimensionadas adequadamente para acomodar um número crescente de entradas sem sacrificar a performance. Isso leva à necessidade de técnicas, como o redimensionamento dinâmico, em que a tabela de *hash* é expandida e re-hashada conforme necessário. Esse processo, embora necessário para manter a eficiência, pode ser computacionalmente caro e deve ser gerenciado cuidadosamente para evitar impactos negativos no desempenho do sistema.

Além dos desafios técnicos, as tabelas de *hash* também apresentam desafios em termos de segurança. Funções de *hash* fracas podem ser vulneráveis a crises, como o ataque de colisão, no qual um adversário gera múltiplas entradas que mapeiam para o mesmo índice de *hash*, degradando o desempenho da tabela. Para sistemas que lidam com dados sensíveis ou que exigem alta confiabilidade, é crucial usar funções de *hash* criptográficas robustas que ofereçam resistência contra tais ataques.

Finalmente, a aplicação prática das tabelas de *hash* em sistemas reais requer uma compreensão profunda do contexto e dos requisitos específicos. Em um banco de dados, por exemplo, uma tabela de *hash* pode ser usada para indexar registros e acelerar buscas, mas deve ser cuidadosamente projetada para lidar com transações concorrentes e garantir a consistência dos dados. Em sistemas de arquivos, as tabelas de *hash* ajudam a gerenciar e localizar arquivos rapidamente, mas devem ser integradas de forma que suportem recuperação eficiente e robustez contra falhas.

VAMOS PRATICAR

1. Em um sistema de gerenciamento de inventário, a rapidez nas operações de busca é essencial para manter a eficiência do sistema. Dessa forma, qual das seguintes estratégias é usada no exemplo de implementação para tratar colisões em uma tabela de *hash*?

   a) *Hashing* duplo.
   b) Endereçamento aberto.
   c) Encadeamento separado.
   d) *Rehashing*.
   e) Árvores AVL

2. Durante uma aula acerca de estruturas de dados, os alunos aprendem a respeito da importância de se escolher uma função de *hash* adequada. Sendo assim, qual das seguintes características é essencial para uma boa função de *hash*?

   a) Alta complexidade.
   b) Distribuição uniforme.
   c) Dependência de ordem.
   d) Utilização de listas encadeadas.
   e) Uso de *hashing* duplo.

3. Em uma aplicação de sistema de *cache*, a escolha da estrutura de dados pode afetar diretamente o desempenho do sistema. Sendo assim, por que uma tabela de *hash* é uma boa escolha para implementar um sistema de cache?

   a) Porque é estável.
   b) Porque tem baixa complexidade de implementação.
   c) Porque permite buscas rápidas.
   d) Porque não utiliza memória adicional.
   e) Porque usa *hashing* duplo.

REFERÊNCIAS

BACKES, A. R. **Algoritmos e estruturas de dados em linguagem C**. Rio de Janeiro: LTC, 2023.

DROZDEK, A. **Estrutura de dados e algoritmos em C++**. São Paulo: Cengage Learning, 2008.

OLIVEIRA, P. M. de; PEREIRA, R. de L. **Estrutura de dados II**. Maringá: Unicesumar, 2019.

PEREIRA, S. do L. **Estruturas de dados em C**: uma abordagem didática. São Paulo: Érica, 2016.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: LTC, 2010.

CONFIRA SUAS RESPOSTAS

1. Alternativa B.

   A opção correta é endereçamento aberto.

2. Alternativa B.

   A opção correta é distribuição uniforme.

3. Alternativa C.

   Uma tabela de hash é uma boa escolha para implementar um sistema de cache, pois permite buscas rápidas.

MEU ESPAÇO

MEU ESPAÇO

MEU ESPAÇO

MEU ESPAÇO

MEU ESPAÇO