

Algoritmos avançados

Prof. Nathan Alves



Objetivos

- Implementar uma estrutura de árvore binária em linguagem C, controlando a inserção, remoção e o armazenamento de objetos do tipo string.
- Manipular uma árvore de busca binária em linguagem C, com a inserção, busca e exibição de objetos, a fim de aplicar os conceitos de ordenação e eficiência na recuperação de dados.
- Organizar de forma integrada as estruturas de árvore binária, árvore de busca e tabela hash, utilizando linguagem C, com a finalidade de desenvolver soluções otimizadas para armazenamento e recuperação de dados inter-relacionados.

Introdução

Olá! Boas-vindas ao desafio de programação, no qual você assumirá o papel de programador(a) responsável pelo desenvolvimento da lógica do jogo Detetive — isso, aquele do Sr. Mostarda com o castiçal na biblioteca! Se você não sabe do que estamos falando, não tem problema! Logo você vai entender como esse jogo é divertido. Já adiantamos: o objetivo é explorar a mansão, encontrar pistas e usar diferentes estratégias para relacioná-las a suspeitos, chegando ao culpado ao final da história.

Para isso, você trabalhará com estruturas de dados, como árvore binária, árvore de busca e tabela hash, cada uma desempenhando um papel específico no jogo. Por exemplo, a árvore binária permitirá ao jogador explorar diferentes cômodos da mansão, a árvore de busca servirá para armazenar e organizar as pistas de forma categórica e a tabela hash permitirá vincular cada pista ao seu suspeito, chegando ao culpado de forma rápida e eficiente (mas sem perder a diversão, é claro!).

A jornada será dividida em três partes, que serão liberadas aos poucos, aumentando a complexidade da história e das responsabilidades de programação. Assim, primeiro, você vai implementar a árvore binária para construir o mapa da mansão, pelo qual o jogador se moverá tentando encontrar diferentes cômodos e descobrir o que ele esconde.

Depois, você vai acrescentar a árvore de busca para armazenar e organizar as pistas. Dessa forma, o jogo consegue inserir, ordenar e exibir o que o detetive já encontrou, tentando dar ao jogador uma visão clara das evidências.

Por fim, você vai integrar a tabela hash. Dessa forma, o jogo consegue vincular cada pista ao suspeito correspondente, auxiliando o detetive a fazer uma avaliação final e dizer, com base nas evidências, quem é o culpado.

Ao longo do caminho, você revisitará conceitos da linguagem C, como structs, ponteiros, alocação de memória e organização de dados, tendo como desafio aplicar tudo na prática para dar vida ao seu próprio jogo.

Prepare-se para pensar como um(a) desenvolvedor(a), tentando encontrar a melhor estrutura de dados para cada situação, aumentando o domínio da programação e a capacidade de abstração. Assim, ao final, você terá criado um sistema de investigação funcional, mostrando como diferentes tipos de dados podem trabalhar juntos para dar forma ao seu próprio jogo.



Neste vídeo, você será responsável por desenvolver a lógica de programação de um jogo de detetive em C, utilizando estruturas como árvore binária, árvore de busca e tabela hash. A cada módulo, novos desafios são incluídos, permitindo explorar a mansão, organizar pistas e identificar o culpado, revisitando conceitos-chave da linguagem.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Antes de navegar e avançar pelos níveis do desafio, vamos conhecer o cenário em que tudo acontece e descobrir qual será a sua missão. Prepare-se!

Cenário

A Enigma Studios, desenvolvedora de jogos voltados para o ensino de lógica e programação, está desenvolvendo um novo título chamado Detective Quest, um jogo eletrônico em que o jogador precisa explorar uma mansão, encontrar pistas e usar diferentes estratégias para relacioná-la a suspeitos.

Para isso, o jogador assume o papel de um detetive encarregado de solucionar o mistério, utilizando estruturas de dados — como árvore binária, árvore de busca e tabela hash — para organizar as informações, registrar o caminho pelo mapa, catalogar as pistas e, ao final, apontar o verdadeiro culpado.

Sua missão

Você foi designado como pessoa desenvolvedora técnica responsável pelo coração do jogo para implementar:

- Uma **árvore binária** para **armazenar o mapa da mansão**, mostrando como ele é formado e como o detetive consegue se mover de um cômodo para o outro.
- Uma **árvore de busca** para **organizar as pistas** de forma categórica, facilitando a recuperação e a exibição de cada nova informação ao jogador.
- Uma **tabela hash** para **vincular cada pista ao seu suspeito**, permitindo que, ao final, o detetive relacione as evidências ao culpado de forma rápida e eficiente.

Seu trabalho é garantir que as **inserções, buscas, remoções e organizações de dados** estejam funcionando como exigido pelo jogo.

Seu código em C deverá gerenciar tais estruturas utilizando:

- Variáveis
- Structs
- Operadores
- Condicionais
- Estruturas aninhadas
- Funções de entrada e saída



Dica

Lembre-se: cada ação do jogador, como mover-se pelo mapa, encontrar uma nova pista ou relacioná-la a um suspeito, deverá ser representada e tratada pelo seu próprio código. Assim, a sua implementação servirá como base para o desenvolvimento das outras mecânicas e para a avaliação final do jogo.

Introdução a árvores binárias

Neste vídeo, apresentamos os fundamentos das árvores binárias, uma estrutura essencial para representar relações hierárquicas. Com exemplos visuais e práticos utilizando strings, exploramos conceitos como nós, raiz, folhas e recursividade. Destacamos aplicações em algoritmos de busca, ordenação e na representação de expressões matemáticas, com implementação em linguagem C.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

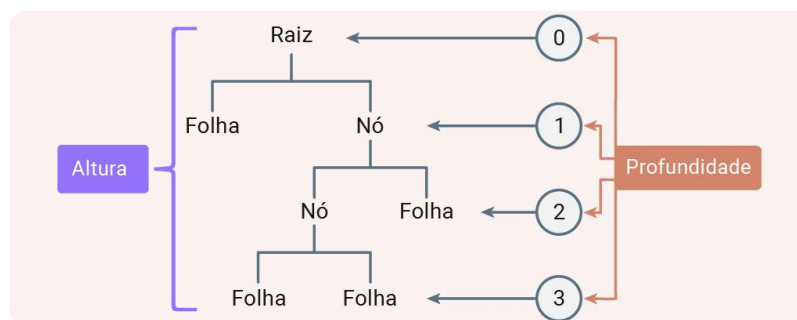
Introdução a árvores binárias e suas características

As estruturas de dados são fundamentais para a ciência da computação, pois permitem a organização eficiente das informações para diversas finalidades. Entre essas estruturas, as árvores binárias se destacam por sua capacidade de representar relações hierárquicas.

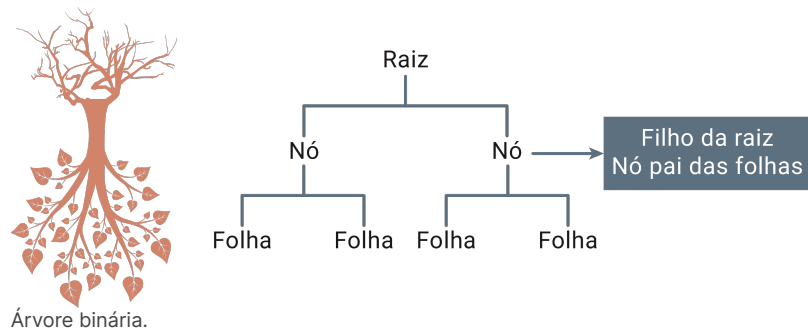
Uma árvore é uma estrutura de dados hierárquica composta por elementos chamados de nós. Cada nó pode conter um valor e ter uma ligação com outros, denominados **filhos**. Se um nó contiver um ou mais filhos, ele se torna um nó **pai**. A árvore binária, por sua vez, é um tipo especial, pois cada nó pai pode ter no máximo dois filhos: o filho à **esquerda** e o filho à **direita**.

O **primeiro** nó de uma árvore é chamado de **raiz**. A partir dele, a estrutura se ramifica, ressaltando que só pode haver uma raiz em cada árvore binária. Os nós que não possuem filhos são chamados de **folhas**. Cada ligação entre nós é chamada de **aresta**, e os caminhos formados por essas conexões ajudam a determinar características importantes da árvore, como a sua altura e a profundidade de seus nós.

Falando nisso, a **profundidade** mede a distância entre a raiz e a última folha selecionado e a **altura** seja, a altura da árvore toda, e a **profundidade**, à distância entre um nó específico

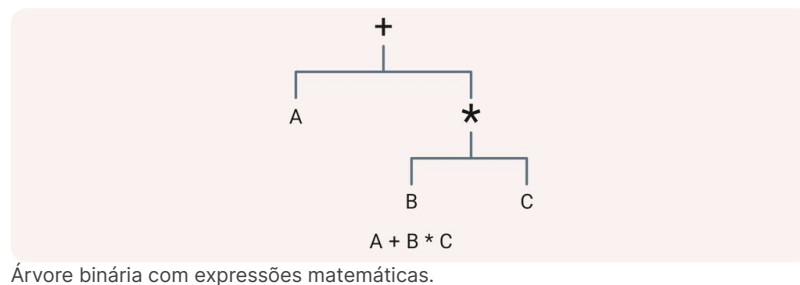


A representação de uma árvore binária se assemelha a uma árvore de cabeça para baixo, com a raiz no topo e os nós descendentes distribuídos em níveis inferiores. Veja, a seguir, a representação de uma árvore binária:



As árvores binárias têm amplo uso na computação: em bancos de dados, compiladores, inteligência artificial e em estruturas de busca e ordenação de dados. Um exemplo clássico é a **árvore binária de busca (BST)**, em que os dados são organizados para viabilizar a busca por elementos de maneira eficiente.

Outra aplicação comum é a **representação de expressões matemáticas**, em que operadores ficam em nós inteiros e operando nas filhas. Isso permite que algoritmos percorram a árvore para calcular expressões com precisão.



Por serem recursivas, as árvores binárias se integram bem a soluções que utilizam **funções recursivas**. Isso facilita operações como inserção, busca e remoção de elementos.

Ao longo deste estudo, vamos explorar desde a teoria até a implementação prática de árvores binárias, utilizando exemplos com strings para facilitar a visualização. Portanto, entender essa estrutura permite compreender variações mais avançadas importantes em diferentes contextos de sistemas computacionais.

Aprofundamento teórico sobre árvores binárias

Neste vídeo, veremos os tipos, propriedades e aplicações das árvores binárias, explorando estruturas como árvores completas, estritas, cheias e de busca. Conhecendo conceitos como altura, profundidade, subárvore e balanceamento, você entenderá como essas estruturas contribuem para a otimização de algoritmos em inteligência artificial, bancos de dados, compiladores e jogos.



Conteúdo interativo

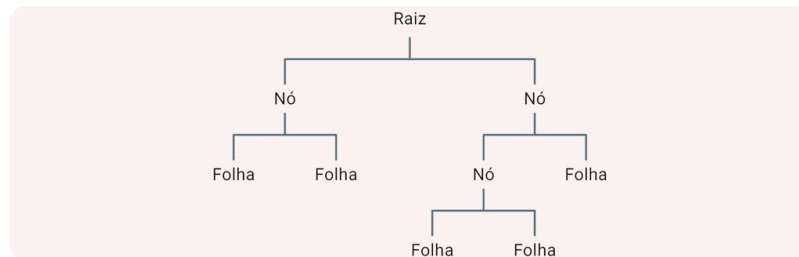
Acesse a versão digital para assistir ao vídeo.

Formas de árvores binárias e suas características

Nas estruturas de dados, poucas formas são tão versáteis quanto as árvores binárias. Elas organizam dados de forma hierárquica, é certo, mas também servem como base para algoritmos eficientes de busca, classificação, entendimento e tomada de decisão. Assim, aprofundando esse conceito, descobrimos variações

e propriedades que tornam as árvores binárias uma ferramenta interessante para representar relações, otimizar operações e resolver problemas complexos de forma elegante.

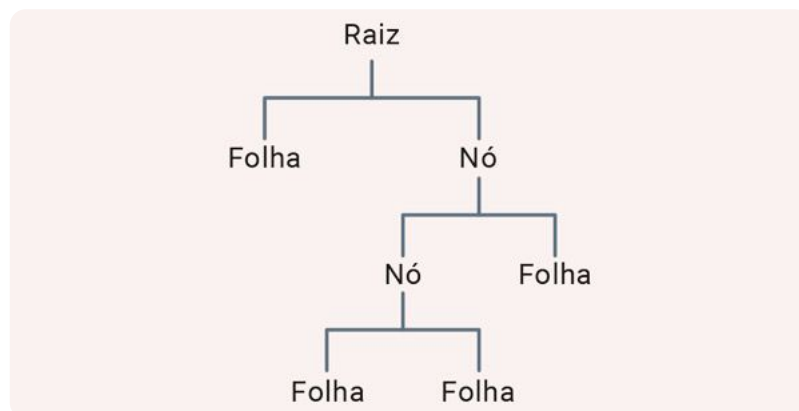
Uma das primeiras distinções entre diferentes árvores binárias está nos **tipos estruturais**. Na **árvore binária completa**, por exemplo, todos os níveis estão preenchidos, com exceção talvez do último, que deve estar mais à esquerda possível.



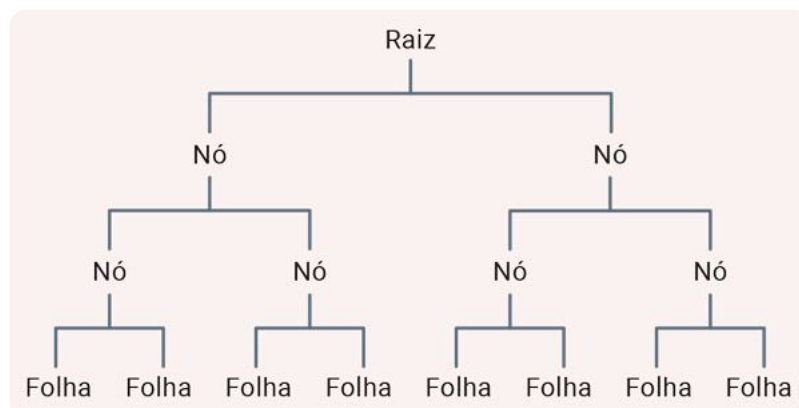
Árvore binária completa.

Já a **árvore estritamente binária** é caracterizada por nós que possuem apenas dois filhos ou nenhum, o que significa que não há nós com apenas um filho. Quando uma árvore é, ao mesmo tempo, completa e cheia, chamamos de **árvore binária cheia**: todos os nós internos têm dois filhos, e todas as folhas estão no mesmo nível.

Veja os casos a seguir!



Árvore estritamente binária.



Árvore binária cheia.

Entre os tipos mais relevantes está a **árvore binária de busca (BST – binary search tree)**, que organiza seus elementos obedecendo a uma regra de ordenação: todos os valores à esquerda de um nó são menores que o valor do próprio nó, e todos os valores à direita são maiores. Essa organização permite realizar operações

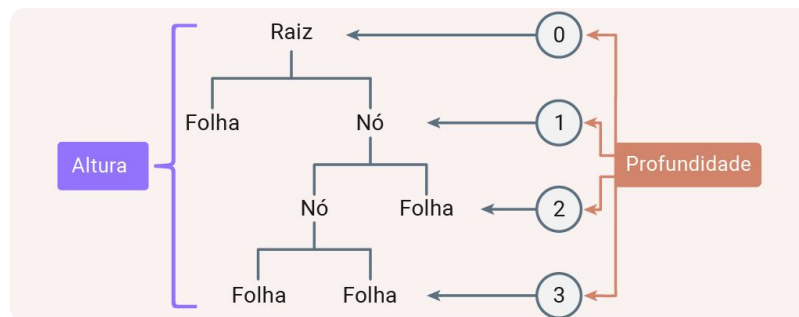
como busca, inserção e remoção com eficiência média de tempo logarítmica, desde que a árvore esteja balanceada.



Comentário

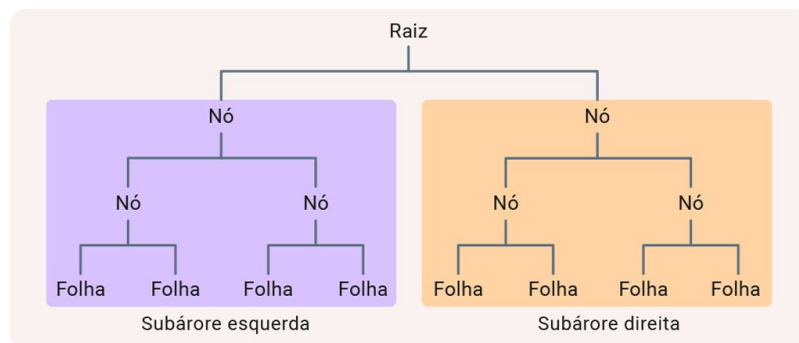
Note que, além dos tipos, as propriedades estruturais também importam. Por exemplo, a altura da árvore, que mede o caminho mais longo de raiz até uma folha, é um dos principais indicadores de eficiência. Tanto é que árvores altas tendem a apresentar pior desempenho em buscas. Por isso precisamos controlar a altura com estratégias de balanceamento.

Outro conceito complementar é a **profundidade de um nó**, que corresponde ao número de arestas entre a raiz e ele. Essa medida ajuda a entender a posição relativa de cada elemento na estrutura, sendo útil, por exemplo, para percursos ou para avaliar o custo de acesso a determinados dados.



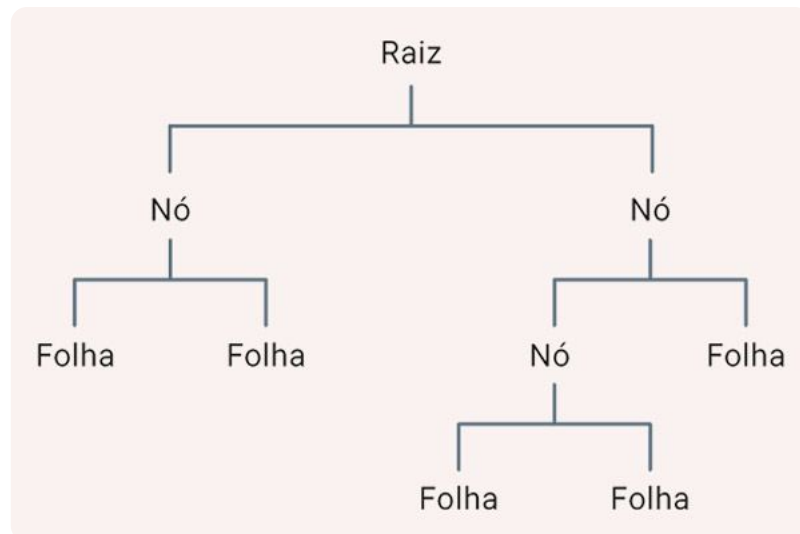
Conceito de altura e profundidade.

Vamos abordar agora o conceito de **subárvore**. Cada nó de uma árvore, junto a seus descendentes, forma uma nova árvore, chamada de subárvore. Em implementações recursivas, operar sobre subárvores é a base de algoritmos eficientes.

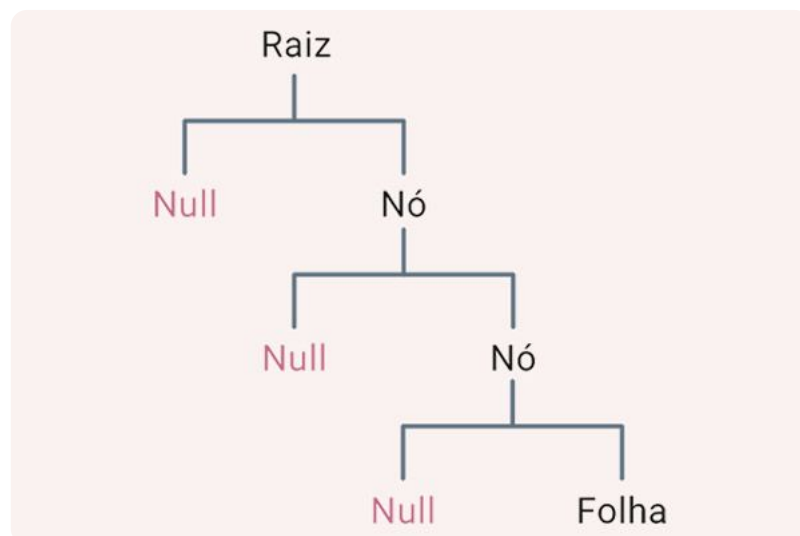


Conceito de subárvore.

A **questão do balanceamento** merece destaque. Uma árvore binária balanceada é aquela cuja altura é mantida próxima do mínimo possível, o que proporciona bom desempenho mesmo para grandes volumes de dados. Em contrapartida, uma árvore desbalanceada pode se degenerar em uma lista encadeada, resultando em complexidades lineares para busca e inserção, que deveriam ser logarítmicas.



Árvore balanceada.



Árvore desbalanceada.

A teoria das árvores binárias não se limita à estrutura em si, pois ela fundamenta inúmeras aplicações práticas na ciência da computação. Em compiladores, por exemplo, árvores de análise sintática representam expressões matemáticas e estruturas de código-fonte. Em bancos de dados, variações como as árvores B e B+ são utilizadas para indexar e buscar registros de forma eficiente.



Curiosidade

No campo da inteligência artificial, árvores de decisão são usadas para classificar dados e automatizar escolhas com base em critérios hierárquicos. Já nos algoritmos de compressão, a árvore binária serve para atribuir códigos de menor comprimento a símbolos mais frequentes, otimizando o espaço necessário para armazenar ou transmitir informações.

Mesmo em contextos lúdicos, como jogos de tabuleiro, de decisão e outras aplicações, a árvore binária se faz presente. Ela organiza as possibilidades futuras de uma partida ou mecânica, permitindo que algoritmos de busca e avaliação determinem a melhor jogada com base nas ramificações do estado atual.

Portanto, compreender, de fato, diferentes tipos, propriedades e aplicações das árvores binárias prepara o terreno para a sua manipulação eficiente e implementação. Ao dominar essa teoria, você terá uma base sólida para criar estruturas de dados robustas, adaptadas a problemas do mundo real e integradas com diversas áreas da computação moderna.

Estrutura de uma árvore binária na prática

Neste vídeo, apresentamos a estrutura básica das árvores binárias em C, abordando a definição de nós com structs, os papéis da raiz, das folhas e dos nós internos, além dos conceitos de altura e profundidade. Com exemplos práticos e códigos explicados passo a passo, você aprenderá a representar e analisar árvores de forma eficiente.

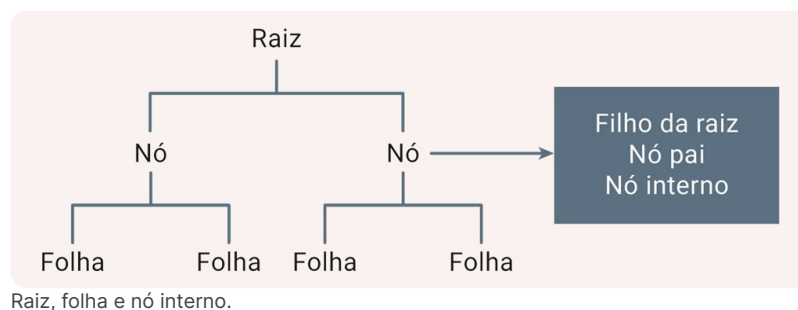


Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Conceitos estruturais de uma árvore binária

Vamos nos aprofundar nos componentes estruturais da árvore binária e no modo como eles são representados e manipulados em código. Entender como definir nós, reconhecer a função da raiz, das folhas e dos nós internos, além de calcular medidas como altura e profundidade, contribui para aplicar essa estrutura em situações reais.



Estrutura de um nó

Para começar a trabalhar com árvores binárias, precisamos de um struct que represente um nó. Veja!

```
c
struct No {
    char valor[50];
    struct No* esquerda;
    struct No* direita;
};
```

Essa estrutura define um nó que armazena uma string e dois ponteiros para os filhos esquerdo e direito. Cada instância de struct No é um elemento da árvore. Lembre-se: essa é a base para todas as operações com árvores binárias.

Por que isso importa?

A estrutura correta do nó viabiliza a construção de árvores de qualquer forma, conectando nós filhos de maneira livre e permitindo chamadas recursivas para percorrer ou modificar a árvore.

Altura da árvore

É o maior número de arestas entre a raiz e uma folha. Em termos práticos, ela define a profundidade máxima da estrutura, e está relacionada ao desempenho de operações como busca.

Confira o código para calcular a altura:

```
c
int altura(struct No* raiz) {
    if (raiz == NULL)
        return 0;
    int alt_esq = altura(raiz->esquerda);
    int alt_dir = altura(raiz->direita);
    return 1 + (alt_esq > alt_dir ? alt_esq : alt_dir);
}
```

Como o código funciona:

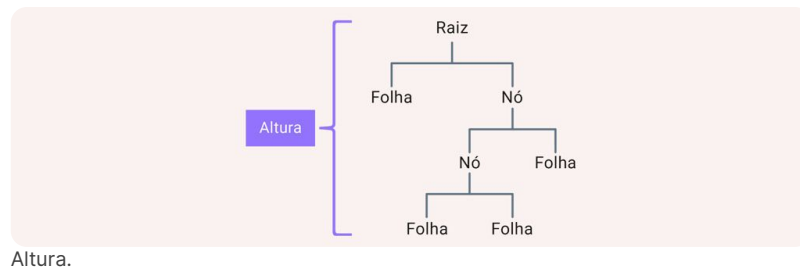
- Se o nó for NULL, a altura é 0, significando que a árvore está vazia ou que chegamos ao fim da árvore.
- Calculamos a altura da subárvore esquerda e direita com chamada recursiva.
 - Comparamos alt_esq e alt_dir, que são as alturas das subárvores esquerda e direita.
 - Usamos o operador ternário (condição ? valor_se_verdadeiro : valor_se_falso) para escolher a maior altura entre as duas.
 - Somamos 1 ao resultado, representando o nível atual da árvore (ou seja, o próprio nó).
- Retornamos a altura da árvore, que é 1 + a maior altura entre as subárvores.



Dica

Identificar a altura de uma árvore é importante, pois afeta o tempo de execução das operações. Assim, uma árvore mais alta pode indicar desbalanceamento, tornando buscas e acessos mais lentos.

Veja como se dá a definição da altura de uma árvore binária!



Altura.

É a distância entre a raiz e um determinado nó. É útil para localizar elementos ou medir o custo de acesso.

Código para calcular a profundidade:

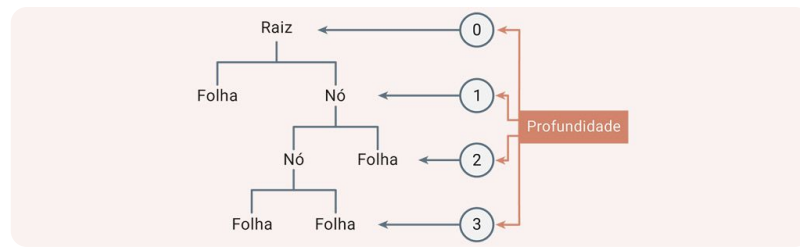
```
c
int profundidade(struct No* raiz, struct No* alvo, int nivel) {
    if (raiz == NULL)
        return -1;
    if (raiz == alvo)
        return nivel;

    int esq = profundidade(raiz->esquerda, alvo, nivel + 1);
    if (esq != -1)
        return esq;

    return profundidade(raiz->direita, alvo, nivel + 1);
}
```

Como o código funciona:

- A função recebe três parâmetros:
 - **Raiz:** o nó atual da árvore binária.
 - **Alvo:** o nó que queremos encontrar.
 - **Nível:** o nível atual da árvore que começa em zero.
- Se a raiz for NULL, significa que atingimos o fim da árvore sem encontrar o nó alvo, então retornamos -1 para indicar que o nó não está presente.
- Se o nó atual (raiz) for o nó alvo, retornamos **nível**, que representa sua profundidade na árvore.
- Chamamos recursivamente a função para a subárvore esquerda, e "**nível + 1**" aumenta a profundidade à medida que descemos na árvore.
- Se encontrarmos o nó alvo na subárvore esquerda, retornamos essa profundidade.
- Se não encontrarmos na esquerda, chamamos recursivamente para a direita, mantendo a lógica.



Profundidade.

Pré-ordem, em ordem e pós-ordem

Neste vídeo, abordamos os três percursos clássicos em árvores binárias: pré-ordem, em ordem e pós-ordem. Com exemplos visuais e implementação em C, você aprenderá como cada abordagem percorre os nós e quando as utilizar, desde clonagem de árvores até ordenação de dados e avaliação de expressões.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O que é um percurso?

Trabalhando com árvores binárias, chega um momento do qual não dá para fugir: percorrê-las. Assim, **percurso** é o processo de visitar todos os nós da árvore de forma sistemática. Isso pode parecer simples à primeira vista, mas existem diversas maneiras de realizar essa travessia, e cada uma delas serve a um propósito diferente. Neste estudo, vamos explorar três abordagens clássicas de percurso: pré-ordem, em ordem e pós-ordem.

Imagine que estamos caminhando por uma árvore, e em cada nó temos três possibilidades: olhar primeiro para ele, depois para seu filho esquerdo e então para o direito. Mas também poderíamos começar pela esquerda, depois o nó atual e, por fim, à direita. Ou quem sabe, deixar o nó para o final, observando primeiro seus dois filhos. Esses três modos de visita correspondem, respectivamente, aos percursos pré-ordem, em ordem e pós-ordem.

Tipos de percurso

Vamos começar vendo detalhadamente como se dá o percurso em uma árvore binária!

Pré-ordem

Neste tipo de percurso, a visita começa pelo próprio nó, seguida pelo filho esquerdo e depois pelo direito. Funciona como um reconhecimento inicial: identificamos o ponto principal antes de explorar seus arredores. É especialmente útil para clonar uma árvore ou salvar sua estrutura original, já que percorre cada elemento na ordem em que está organizado.

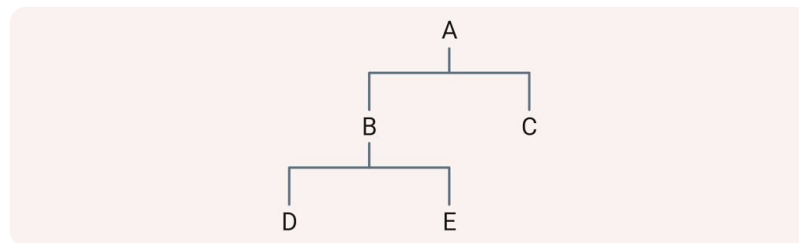
Em ordem

Amplamente utilizado em árvores binárias de busca, este percurso segue a sequência: filho esquerdo, nó atual e filho direito. Essa organização resulta em uma travessia com elementos dispostos em ordem crescente, sendo ideal para buscas eficientes, listagens ordenadas e aplicações que exigem hierarquia clara entre os dados.

Pós-ordem

Aqui, o nó atual é deixado por último. Primeiro visitamos o filho esquerdo, depois o direito, e, por fim, retornamos ao nó principal. Essa abordagem é indicada para desmontar a árvore com segurança, liberar memória ou avaliar expressões aritméticas compostas, já que processa os componentes antes de lidar com o ponto central.

Para ilustrar, considere esta árvore:



Árvore binária básica.

Se percorrermos a árvore em **pré-ordem**, a sequência será A, B, D, E, C. Já em **em ordem**, teremos D, B, E, A, C. No caso da **pós-ordem**, a ordem será D, E, B, C, A.

Implementação de percursos

Cada um dos percursos vistos pode ser implementado de forma recursiva em C. A função **preOrdem**, por exemplo, começa imprimindo o valor do nó atual, depois chama a si mesma para o filho esquerda e, em seguida, para o direito. A **emOrdem** faz o mesmo, mas imprime o valor entre as duas chamadas recursivas. Já a **posOrdem** só imprime o valor do nó depois de visitar ambos os filhos. Acompanhe!

```

c

void preOrdem(struct No* raiz) {
    if (raiz != NULL) {
        printf("%s ", raiz->valor);
        preOrdem(raiz->esquerda);
        preOrdem(raiz->direita);
    }
}

void emOrdem(struct No* raiz) {
    if (raiz != NULL) {
        emOrdem(raiz->esquerda);
        printf("%s ", raiz->valor);
        emOrdem(raiz->direita);
    }
}

void posOrdem(struct No* raiz) {
    if (raiz != NULL) {
        posOrdem(raiz->esquerda);
        posOrdem(raiz->direita);
        printf("%s ", raiz->valor);
    }
}

```

Implementação de uma árvore binária

Neste vídeo, você verá como implementar uma árvore binária completa em C, usando strings como conteúdo. A partir da criação de nós com alocação dinâmica até os percursos pré-ordem, em ordem e pós-ordem, o conteúdo consolida conceitos como ponteiros, recursividade e liberação de memória, preparando o terreno para estruturas mais avançadas.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Implementando a árvore binária

Vamos agora implementar a árvore binária de forma completa, utilizando strings como conteúdo de cada nó. Este exercício é ideal para consolidar os conceitos de ponteiros, alocação dinâmica e recursividade.

Definindo a estrutura e criação do nó

Começamos com a definição do nó. Ele conterá uma string (com até 49 caracteres + \0) e dois ponteiros: um para o filho à esquerda e outro para o que está à direita.

```
c
#include
#include
#include

struct No {
    char valor[50];
    struct No* esquerda;
    struct No* direita;
};
```

Neste exemplo, vamos definir, de modo manual, nossa árvore. Então, não faremos a função **inserir**. Desse modo, basta criamos uma função para alocar um novo nó na memória:

```
c
struct No* criarNo(char* valor) {
    struct No* novo = (struct No*) malloc(sizeof(struct No));
    strcpy(novo->valor, valor);
    novo->esquerda = NULL;
    novo->direita = NULL;
    return novo;
}
```

Definindo percursos

Depois que a árvore for construída, podemos percorrê-la. Aqui estão todos os percursos para ficar mais clara a visualização do resultado de cada um deles:

```
c
void preOrdem(struct No* raiz) {
    if (raiz != NULL) {
        printf("%s ", raiz->valor);
        preOrdem(raiz->esquerda);
        preOrdem(raiz->direita);
    }
}

void emOrdem(struct No* raiz) {
    if (raiz != NULL) {
        emOrdem(raiz->esquerda);
        printf("%s ", raiz->valor);
        emOrdem(raiz->direita);
    }
}

void posOrdem(struct No* raiz) {
    if (raiz != NULL) {
        posOrdem(raiz->esquerda);
        posOrdem(raiz->direita);
        printf("%s ", raiz->valor);
    }
}
```

Liberação e utilização no main

É preciso liberar a memória ocupada pela árvore após o uso, evitando vazamentos:

```
c
void liberar(struct No* raiz) {
    if (raiz != NULL) {
        liberar(raiz->esquerda);
        liberar(raiz->direita);
        free(raiz);
    }
}
```

Por fim, tudo isso se une em um programa completo. No **main**, vamos criar a árvore e inserir algumas strings:

```
c
int main() {
    struct No* raiz = criarNo("Hall de Entrada");
    raiz->esquerda = criarNo("Sala de Estar");
    raiz->direita = criarNo("Biblioteca");
    raiz->esquerda->esquerda = criarNo("Quarto");

    printf("Pré-ordem: ");
    preOrdem(raiz);
    printf("\n");

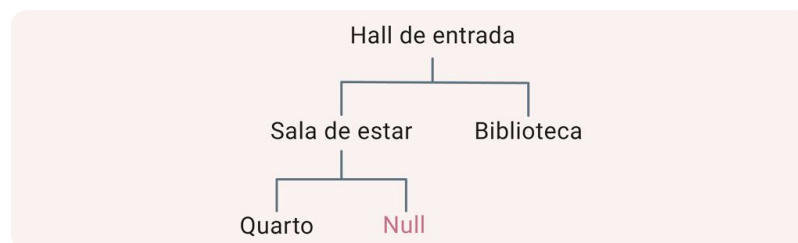
    printf("Em ordem: ");
    emOrdem(raiz);
    printf("\n");

    printf("Pós-ordem: ");
    posOrdem(raiz);
    printf("\n");

    liberar(raiz);
    return 0;
}
```

Saída para cada percurso

Veja a estrutura gerada na árvore criada no **main**:



Estrutura de cômodos em árvore.

Veja a seguir como é estruturada a saída do programa:

Pré-ordem

Hall de entrada Sala de estar Quarto Biblioteca

Em ordem

Quarto Sala de estar Hall de entrada Biblioteca

Pós-ordem

Quarto Sala de estar Biblioteca Hall de entrada

As três formas de percorrer a árvore revelam diferentes lógicas de navegação:

- **Pré-ordem** visita primeiro o nó raiz, depois os filhos à esquerda e à direita.
- **Em ordem** percorre da esquerda até a raiz, e depois vai para a direita — sendo útil para ordenações quando aplicadas em árvores de busca.
- **Pós-ordem** visita os filhos antes de tratar a raiz, sendo comum em liberações de memória ou análise de dependências.

Hora de codar

Neste vídeo prático, você construirá uma árvore binária simples em C, definindo structs, alocação dinâmica e percursos fundamentais. Essa base é crucial para montar interativamente o mapa da mansão em Detective Quest, ligando teoria e prática antes do desafio de exploração.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Você irá desenvolver uma árvore binária em linguagem C, utilizando alocação dinâmica de memória para definir sua estrutura e armazenar os dados de forma eficiente. Prepara-se!

Etapas do desenvolvimento

1. Definição da estrutura da árvore binária, com os campos necessários para armazenar os dados e os ponteiros para os nós esquerdo e direito.
2. Implementação da inserção de elementos utilizando alocação dinâmica.
3. Exploração dos percursos (caminhamentos) da árvore:
 1. Pré-ordem
 2. Em ordem
 3. Pós-ordem

Desafio: nível novato

Neste vídeo de desafio, você aplicará a árvore binária para criar o mapa interativo da mansão em Detective Quest. Implemente funções `criarSala()`, `explorarSalas()` e navegue do Hall de entrada até um nó-folha, garantindo usabilidade, legibilidade e documentação clara.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O que você vai fazer?

A Enigma Studios, desenvolvedora de jogos voltados para o ensino de lógica e programação, está desenvolvendo um novo título chamado **Detective Quest**, um jogo eletrônico em que o jogador precisa explorar uma mansão para encontrar o culpado.

Designado como **desenvolvedor(a) técnico(a)**, você é responsável por implementar o sistema que controla o mapa da mansão, representado por uma **árvore binária**. O seu trabalho será garantir que o mapa seja criado com exatidão e que permita ao jogador explorá-lo a partir do Hall de entrada, escolhendo entre os caminhos à esquerda ou à direita em cada sala.

Você implementará um programa em **C** que simula o mapa da mansão como uma árvore binária com **nome para cada cômodo**. A árvore é montada de modo automático em alocação dinâmica, e o jogador poderá explorar esse mapa até chegar a um cômodo que não tenha mais caminhos.

Requisitos funcionais

Seu programa em C deverá:

- Criar uma **árvore binária** para representar o mapa da mansão.
- Permitir a **exploração interativa** da mansão a partir do Hall de entrada, escolhendo ir para a esquerda (e) ou para a direita (d).

Além disso, a **estrutura da mansão já vem definida no código**, e não é necessário inseri-la manualmente — afinal, ela é criada de modo automático pela função `main()`, usando a função `criarSala()`.

Seu programa em C ainda deverá viabilizar a **exploração continua** até o jogador alcançar um cômodo que não possua caminhos à esquerda nem à direita (isto é, um **nó-folha** na árvore). Por fim, o programa **exibe o nome de cada sala visitada** durante a exploração.

Cada cômodo possui:

- **nome**: uma string que identifica a sala (por exemplo: Sala de estar, Cozinha ou Jardim).

Requisitos não funcionais

- **Usabilidade**: a saída do programa deve ser clara e direta, guiando o jogador na exploração.
- **Legibilidade**: o código deve estar bem-organizado, com nomes intuitivos e identificação apropriada.

- **Documentação:** o código deve possuir comentários explicando o propósito das funções, como:
 - `criarSala()` – cria, de forma dinâmica, uma sala com nome.
 - `explorarSalas()` – permite a navegação do jogador pela árvore.
 - `main()` – monta o mapa inicial e dá início à exploração.

Simplificações para o nível novato

- Utiliza apenas **árvore binária** (sem busca, inserção dinâmica ou remoção).
- Há um **menu de opções**; a navegação acontece a partir das escolhas do usuário (e, d ou s para sair).
- Cria-se a árvore de forma manual no código-fonte, e ela **não é modificada** em tempo de execução.

Conceitos trabalhados

- **Árvore binária:** estrutura de dados hierárquica com dois filhos por nó.
- **Structs:** Criação de um tipo personalizado (Sala) com campos para nome e ponteiros.
- **Alocação dinâmica:** uso de `malloc` para criação de nós da árvore.
- **Operadores condicionais:** controle das decisões do jogador (if, else).
- **Modularização:** separação de funcionalidades em funções distintas, com responsabilidades claras.

Entregando seu projeto

1. **Desenvolva seu projeto no GitHub:** use o mesmo repositório do GitHub dos níveis anteriores.
2. **Atualize o arquivo do seu código:** edite o arquivo principal do seu projeto, inserindo o código completo já com as novas funcionalidades.
3. **Compile e teste:** faça isso com rigor, garantindo que todas as comparações e cálculos estejam corretos.
4. **Faça commit e push:** faça commit das suas alterações e envie (push) para o seu repositório no GitHub.
5. **Envie o link do repositório no GitHub:** faça isso através da plataforma SAVA.

Tutorial git

Você está prestes a aplicar os conceitos aprendidos para resolver um desafio prático no ambiente do GitHub. Veja as instruções gerais a seguir para acessar, aceitar e executar o desafio, de modo que sua solução esteja bem estruturada e documentada.

Dê o primeiro passo

Acesse o GitHub Classroom. Nesse ambiente, você terá acesso ao repositório padrão do desafio.

Caso ainda não tenha uma conta no GitHub, não se preocupe: você pode criar uma grátis, clicando no [link](#).

Aceite o desafio

Tenha acesso ao repositório no GitHub, no qual você encontrará o repositório criado para o desenvolvimento do seu desafio.

Acesse o repositório

Clique no link do repositório para abrir o ambiente GitHub com a descrição do desafio e a estrutura modelo de arquivos e pastas que deve ser utilizada. Lembre-se: é esse o link que você deve enviar no SAVA.

Explore a estrutura do ambiente

Veja a estrutura organizada de pastas e arquivos necessários para o desenvolvimento do desafio.

Desenvolva o desafio

Utilize o GitHub CodeSpace para editar o arquivo do código-fonte e desenvolver o desafio. Certifique-se de que o código esteja organizado e funcional para resolver o problema proposto.

Entregue o desafio

Forneça o repositório do GitHub com todos os arquivos de código-fonte e conteúdos relacionados ao projeto. Certifique-se de que o repositório esteja bem estruturado, com pastas e arquivos nomeados de maneira clara e coerente. Envie o link para o repositório do seu desafio no GitHub.

Finalizando nosso estudo, aqui está a última orientação: comente todos os arquivos de código-fonte, pois isso demonstra o quanto você sabe sobre o funcionamento do código e facilita a correção por terceiros. Seus comentários devem explicar a finalidade das principais seções do código, o funcionamento de algoritmos complexos e o propósito de variáveis e funções utilizadas.

Assista agora ao último vídeo, com dicas e detalhes sobre a entrega do seu projeto. Até a próxima!



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Introdução a árvores binárias de busca e tries

Neste vídeo, introduzimos as árvores BST e Trie, destacando regras de ordenação, balanceamento automático e organização por prefixos. Você compreenderá seus principais cenários de uso e diferenças estruturais, preparando-se para implementá-las em linguagem C no contexto de coleta e organização de pistas no jogo Detective Quest.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Introdução às estruturas: BST e Tries

Organizar dados de forma eficiente e com acesso rápido é uma necessidade recorrente em diversas áreas da computação. Diante disso, neste estudo, abordaremos três estruturas de dados que cumprem esse papel com diferentes estratégias: as **árvores binárias de busca (BST)** e as **Tries**. Embora cada uma possua regras e finalidades distintas, todas compartilham o princípio de representar dados de maneira hierárquica para otimizar operações como busca, inserção e exclusão. Vamos lá!

1

Árvore binária de busca (BST)

Provavelmente a mais clássica das árvores de busca. Cada nó contém um valor e até dois filhos. A principal regra da BST estabelece que todos os valores menores que o de um nó ficam à esquerda, e os maiores, à direita. Essa ordenação natural permite que buscas por elementos ocorram de forma eficiente e com complexidade média de tempo $O(\log n)$, mas desde que a árvore esteja balanceada. No entanto, lembre-se: a eficiência da BST depende da forma como os dados são inseridos. Em casos desfavoráveis, como inserção de dados em ordem crescente, a estrutura pode se degradar para algo próximo a uma lista encadeada, perdendo sua principal vantagem.

2

Trie

Atuam em um campo distinto, porém igualmente relevante. A trie, também conhecida como árvore de prefixos, difere da BST, que opera com comparações entre valores. Ela organiza strings caractere por caractere: cada nível representa uma posição na string, enquanto cada ramificação indica uma possibilidade de caractere. Assim, palavras com prefixos comuns compartilham parte do caminho dentro da estrutura. Essa organização permite buscas extremamente rápidas por palavras completas e prefixos, com complexidade proporcional apenas ao tamanho da palavra, independentemente da quantidade de elementos armazenados.

Dica: tries são amplamente utilizadas em sistemas de autocompletar, corretores ortográficos e algoritmos de compressão.

Embora possuam finalidades diferentes, as duas estruturas que estudaremos aqui (**BST e Trie**) oferecem soluções interessantes para problemas comuns em ciência da computação. Assim, a **BST** será explorada como a base para entendermos árvores de busca e sua implementação prática. Em seguida, a **Trie** será estudada como uma abordagem alternativa e bastante eficaz para manipulação de strings.

Árvores binárias de busca (BST)

Neste vídeo, você aprenderá a implementar uma árvore binária de busca em linguagem C. Serão abordados conceitos como a definição de structs, implementação da função de inserção recursiva e o percurso em ordem para exibir strings organizadas alfabeticamente. Além disso, você verá como realizar buscas de forma eficiente, reforçando a importância dessa estrutura para a manipulação de dados.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Fundamentos da BST (binary search tree)

A **árvore binária de busca**, conhecida como **BST (binary search tree)**, é uma estrutura importante para organizar dados de forma hierárquica. Seu principal objetivo é acelerar a busca de informações, respeitando uma regra simples: em cada nó, todos os elementos à esquerda são **menores** que o valor atual, enquanto os à direita são **maiores**. Isso permite que, a cada comparação, metade da árvore seja descartada, o que torna as operações muito mais eficientes do que em estruturas lineares (por exemplo, listas).

Diante disso, vamos aprender e implementar uma BST, trabalhando com **strings** como valores. Mais do que escrever o código, a intenção é entender **como** ele funciona e **por que** cada parte é necessária. Em seguir, vamos construir a árvore por etapas e, ao final, visualizar sua estrutura e saída no console.

Estrutura do nó

Primeiro, é necessário definir a estrutura do nó. Lembrando: cada nó possui dois ponteiros para seus filhos e um campo para o valor armazenado. A diferença aqui é que usaremos **strings**. Veja!

```
c

#include
#include
#include

struct No {
    char valor[50];
    struct No* esquerda;
    struct No* direita;
};
```

Cada nó possui três componentes:

- **valor:** o dado armazenado (nesse caso, uma string).
- **esquerda:** ponteiro para o filho à esquerda.
- **direita:** ponteiro para o filho à direita.



Dica

Essa estrutura é a base da nossa árvore e servirá para construir, percorrer e modificar os dados.

Criando um nó

Sempre que quisermos adicionar um novo valor à árvore, precisaremos criar um nó com esse valor. Essa função aloca memória e inicializa os ponteiros como NULL, já que novos nós começam sem filhos. Acompanhe!

```
c
struct No* criarNo(const char* valor) {
    struct No* novo = (struct No*) malloc(sizeof(struct No));
    strcpy(novo->valor, valor);
    novo->esquerda = NULL;
    novo->direita = NULL;
    return novo;
}
```

Nesse momento, é bom que notemos o seguinte:



Atenção

Esse passo certifica que cada valor inserido é um novo bloco na árvore, isolado e pronto para ser conectado a outros nós.

Inserindo elementos na árvore

O passo agora é inserir os nós seguindo a regra da BST: valores menores vão à esquerda, maiores à direita:

```
c
struct No* inserir(struct No* raiz, const char* valor) {
    if (raiz == NULL)
        return criarNo(valor);

    if (strcmp(valor, raiz->valor) < 0)
        raiz->esquerda = inserir(raiz->esquerda, valor);
    else if (strcmp(valor, raiz->valor) > 0)
        raiz->direita = inserir(raiz->direita, valor);

    return raiz;
}
```

Por último, o processo estudado é recursivo, ou seja, ele percorre a árvore até encontrar uma posição livre para inserir o novo valor. Como cada chamada vai descendo um nível, a árvore cresce de cima para baixo. Toda vez que um nó for inserido, é considerada a lógica que caracteriza uma BST. Quando um nó é menor que a raiz atual, ele é alocado à esquerda dela, caso contrário, à direita.

Percorrendo a árvore em ordem (in-order)

A principal vantagem da BST é poder ser percorrida em ordem crescente com um algoritmo simples. Esse percurso visita primeiro a esquerda, depois o nó atual, e, por fim, a direita. Veja!

```
c
void emOrdem(struct No* raiz) {
    if (raiz != NULL) {
        emOrdem(raiz->esquerda);
        printf("%s ", raiz->valor);
        emOrdem(raiz->direita);
    }
}
```

Temos, então, um ponto que merece ser visto. Note:



Relembrando

O tipo de travessia visto é ideal para exibir os valores organizados de forma alfabética ou numérica.

Buscando um valor na árvore

Procurar em uma BST é muito mais eficiente do que em uma lista, pelo simples fato de que podemos eliminar metade dos elementos a cada comparação. Observe:

```
c
int buscar(struct No* raiz, const char* chave) {
    if (raiz == NULL)
        return 0;

    if (strcmp(chave, raiz->valor) == 0)
        return 1;
    else if (strcmp(chave, raiz->valor) < 0)
        return buscar(raiz->esquerda, chave);
    else
        return buscar(raiz->direita, chave);
}
```

O algoritmo segue a lógica da inserção: se o valor é menor, buscamos à esquerda. Se maior, à direita. Quando encontramos, retornamos 1; caso contrário, 0.

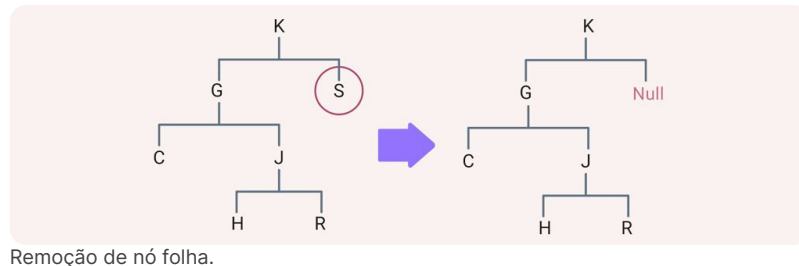
Remoção de um nó

É um processo complexo, pois não se pode remover o nó de uma árvore binária sem considerar os casos distintos para exclusão. São eles:

Remoção na folha

A exclusão de um nó folha ocorre quando o nó a ser excluído não possui filhos. Nesse caso, o procedimento é simples: o ponteiro do pai é ajustado para NULL e o nó é liberado da memória. Como não há subárvores envolvidas, a estrutura da árvore se mantém válida e não é nenhum reordenamento adicional.

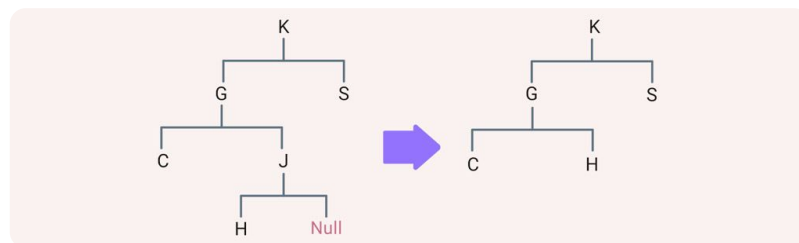
Confira na imagem a seguir!



Remoção de nó folha.

Remoção de um nó com um filho

Se for um único filho, a exclusão exige que o ponteiro do pai do nó removido passe a apontar para o filho existente, que pode estar à esquerda ou à direita. Isso permite que os elementos inferiores continuem acessíveis e preserva a ordenação da árvore binária de busca. O nó retirado é, então, liberado da memória sem quebrar a hierarquia da árvore. Veja!



Remoção de nó com um filho.

Remoção de nó com dois filhos

Em uma árvore binária de busca, a retirada de um nó com dois filhos exige substituição cuidadosa para manter a ordenação da estrutura. Existem duas abordagens corretas e equivalentes:

Usar o sucessor

Localizar o menor valor da subárvore direita do nó a ser removido. Esse valor é o primeiro maior do que o valor atual. Após copiar esse valor para o nó original, remova o nó que continha esse sucessor na subárvore.



Usar o antecessor

Identificar o maior valor da subárvore esquerda do nó a ser removido. Esse valor é o último menor do que o valor atual. Assim como no caso anterior, ele é copiado para o nó original, que é, então, removido.

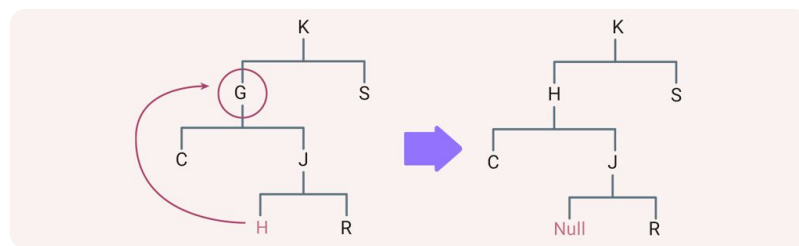
Nesse ponto temos algo importante a observar. Veja!



Atenção

Ambas as alternativas mantêm a propriedade fundamental da árvore binária de busca, isto é, todos os valores à esquerda de um nó são menores que ele, e todos à direita são maiores. A escolha entre sucessor ou antecessor fica a critério da implementação.

Veja, na imagem a seguir, como se dá a remoção de um nó com dois filhos!



Remoção do nó com dois filhos.

Em seguida, verificamos se a árvore está vazia. Se sim, ou se a chamada recursiva chegou a um nó inexistente, retorna NULL.

```
c
if (raiz == NULL)
    return NULL;
```

Depois, navegamos até o nó com o valor a ser removido. Comparando a string **valor** com o conteúdo do nó atual, se for menor, o valor vai para a subárvore esquerda; se for maior, para a direita. A busca é feita de forma recursiva até encontrar o valor exato.

```
c
if (strcmp(valor, raiz->valor) < 0) {
    raiz->esquerda = remover(raiz->esquerda, valor);
} else if (strcmp(valor, raiz->valor) > 0) {
    raiz->direita = remover(raiz->direita, valor);
}
```

Se não for menor, nem maior, achamos nosso valor, entrando em:

```
c
else {
```

Caso 1: nó sem filhos (folha). Remove o nó e retorna NULL para seu pai. Confira:

```
c
if (raiz->esquerda == NULL && raiz->direita == NULL) {
    free(raiz->valor); // Libera a string alocada dinamicamente
    free(raiz);
    return NULL;
}
```

Caso 2: nó com um único filho. Substituímos o nó pelo filho correspondente:

```
c
else if (raiz->esquerda == NULL) {
    struct No* temp = raiz->direita;
    free(raiz->valor);
    free(raiz);
    return temp;
} else if (raiz->direita == NULL) {
    struct No* temp = raiz->esquerda;
    free(raiz->valor);
    free(raiz);
    return temp;
}
```

Caso 3: nó com dois filhos. Encontra a menor string da subárvore direita (ordem alfabética), copia para o nó atual e remove o nó duplicado de maneira recursiva. A função **strdup()** cria uma cópia da string.

```
c
else {
    struct No* temp = raiz->direita;
    while (temp->esquerda != NULL)
        temp = temp->esquerda;

    free(raiz->valor);
    raiz->valor = strdup(temp->valor); // Copia a string do sucessor
    raiz->direita = remover(raiz->direita, temp->valor);
}
```

Versão final da função de remover:

c

```
struct No* remover(struct No* raiz, const char* valor) {
    if (raiz == NULL)
        return NULL;

    if (strcmp(valor, raiz->valor) < 0) {
        raiz->esquerda = remover(raiz->esquerda, valor);
    } else if (strcmp(valor, raiz->valor) > 0) {
        raiz->direita = remover(raiz->direita, valor);
    } else {
        // Caso 1: sem filhos
        if (raiz->esquerda == NULL && raiz->direita == NULL) {
            free(raiz->valor);
            free(raiz);
            return NULL;
        }

        // Caso 2: um único filho
        else if (raiz->esquerda == NULL) {
            struct No* temp = raiz->direita;
            free(raiz->valor);
            free(raiz);
            return temp;
        } else if (raiz->direita == NULL) {
            struct No* temp = raiz->esquerda;
            free(raiz->valor);
            free(raiz);
            return temp;
        }

        // Caso 3: dois filhos
        else {
            struct No* sucessor = encontrarMinimo(raiz->direita);
            free(raiz->valor);
            raiz->valor = strdup(sucessor->valor);
            raiz->direita = remover(raiz->direita, sucessor->valor);
        }
    }

    return raiz;
}
```

Sobre essa versão final da função remover podemos destacar o seguinte:



Dica

O código visto cobre todos os casos de maneira certa e permite que a memória alocada para as strings também seja liberada quando um nó é removido.

Função principal (main)

Montamos a árvore, imprimimos os valores ordenados e testamos a busca de alguns nomes. Acompanhe!

```

c
int main() {
    struct No* raiz = NULL;

    raiz = inserir(raiz, "Pegadas de Lama");
    raiz = inserir(raiz, "Chave perdida");
    raiz = inserir(raiz, "Livro com página faltando");
    raiz = inserir(raiz, "Lençol manchado");
    raiz = inserir(raiz, "Gaveta perdida");

    printf("\nElementos em ordem: ");
    emOrdem(raiz);

    printf("\nBusca por 'Lençol manchado': %s", buscar(raiz, "Lençol manchado") ?
    "Encontrado" : "Não encontrado");
    printf("\nBusca por 'Óculos': %s\n", buscar(raiz, "Óculos") ? "Encontrado" : "Não
    encontrado");

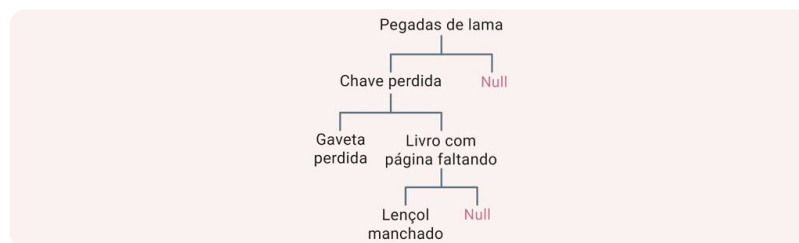
    return 0;
}

```

O operador ternário (condição ? ValorSeVerdadeiro: valorSeFalso) é utilizado para exibir “Encontrado” caso a função **buscar()** retorne verdadeira, indicando que o item foi localizado na árvore. Se retornar falso, o valor exibido será “Não encontrado”. Dica: essa técnica é muito utilizada para simplificar decisões lógicas dentro do printf.

Com isso, temos uma BST funcional, capaz de armazenar, ordenar e buscar strings de maneira eficiente.

Diagrama



Estrutura da árvore de pistas.

Saída

Elementos em ordem: Chave perdida Gaveta perdida Lençol manchado Livro com página faltando Pegadas de Lama

Busca por 'Lençol manchado': Encontrado

Busca por 'Óculos': Não encontrado

Árvores Trie

Neste vídeo, você verá como implementar uma trie em C, criando nós com arrays de ponteiros para cada caractere e marcação de fim de palavra. Além disso, verá funções de inserção e busca por prefixo.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Introdução a árvores Trie

Quando falamos de árvore Trie, precisamos saber que a pronúncia correta é “traí” e que é também conhecida como **árvore de prefixos**. Ela é uma estrutura de dados especializada no armazenamento e manipulação de **strings**, com excelente desempenho para buscas rápidas, autocompletar e verificação de palavras. Ao contrário das árvores BST, que organizam os dados de acordo com valores numéricos ou ordem alfabética total, a Trie organiza cada palavra **letra por letra**, aproveitando ao máximo os **prefixos comuns** entre elas.

Essa estrutura é bastante utilizada em dicionários digitais, sistemas de correção automática, algoritmos de compressão e qualquer aplicação que envolva grande quantidade de palavras ou buscas frequentes com base em prefixos.

Estrutura do nó

Para construir uma Trie, cada nó precisa representar possíveis ramificações para os próximos caracteres. Em geral, utilizamos um array de ponteiros (por exemplo, 26 para letras minúsculas de 'a' a 'z') e um indicador booleano que mostra se um caminho forma uma palavra completa. Veja!

```
c
#include
#include
#include
#include

#define TAMANHO_ALFABETO 26

struct NoTrie {
    struct NoTrie* filhos[TAMANHO_ALFABETO];
    bool ehFimDePalavra;
};
```

A estrutura reserva espaço para até 26 possíveis filhos em cada nível e um campo ehFimDePalavra, que nos ajuda a reconhecer quando uma sequência de caracteres completa uma palavra.

Criação de um nó

Ao iniciarmos a Trie, precisaremos de um nó raiz vazio. Cada novo caractere inserido será conectado de modo dinâmico, conforme a palavra for processada.

```
c
struct NoTrie* criarNo() {
    struct NoTrie* novoNo = (struct NoTrie*) malloc(sizeof(struct NoTrie));
    novoNo->ehFimDePalavra = false;
    for (int i = 0; i < TAMANHO_ALFABETO; i++) {
        novoNo->filhos[i] = NULL;
    }

    return novoNo;
}
```

A alocação dinâmica permite que o espaço de memória seja utilizado apenas quando necessário. Cada ponteiro inicializado com NULL viabiliza que a árvore cresça sob demanda, economizando espaço.

Inserção de palavras na Trie

Percorremos caractere por caractere e criamos os nós intermediários se eles ainda não existirem. A marcação de fim de palavra é essencial para distinguir palavras completas de simples prefixos. Confira:

```
c
void
(struct NoTrie* raiz, const char* palavra) {
    struct NoTrie* atual = raiz;

    for (int i = 0; palavra[i] != '\0'; i++) {
        int indice = palavra[i] - 'a';

        if (atual->filhos[indice] == NULL) {
            atual->filhos[indice] = criarNo();
        }
        atual = atual->filhos[indice];
    }
    atual->ehFimDePalavra = true;
}
```

O uso de `palavra[i] - 'a'` transforma letras em índices de 0 a 25. Assim, a letra 'c' vira índice 2, por exemplo. Isso permite acesso direto ao filho correspondente, sem necessidade de laços ou comparações.

Busca de palavras

É muito parecida com a inserção, ou seja, percorremos os nós conforme os caracteres da palavra. Se em algum momento o caminho não existir, significa que a palavra não está presente. E, mesmo se todos os caracteres forem encontrados, o nó final precisa estar marcado como fim de palavra, conforme vemos a seguir:

```
c
bool buscar(struct NoTrie* raiz, const char* palavra) {
    struct NoTrie* atual = raiz;

    for (int i = 0; palavra[i] != '\0'; i++) {
        int indice = palavra[i] - 'a';

        if (atual->filhos[indice] == NULL)
            return false;

        atual = atual->filhos[indice];
    }
    return atual != NULL && atual->ehFimDePalavra;
}
```

A função retorna true apenas se a sequência existir e estiver marcada como palavra completa. Isso evita confundir prefixos com palavras reais.

Ordenação lexicográfica e normalização

Como a Trie armazena os caracteres em ramos que seguem a ordem do alfabeto (de 'a' a 'z'), podemos percorrê-la em pré-ordem alfabética para imprimir todas as palavras de forma ordenada. Esse percurso pode ser feito de modo recursivo.

```

c

void listarPalavras(struct NoTrie* no, char* buffer, int nivel) {
    if (no->ehFimDePalavra) {
        buffer[nivel] = '\0';
        printf("%s\n", buffer);
    }

    for (int i = 0; i < TAMANHO_ALFABETO; i++) {
        if (no->filhos[i] != NULL) {
            buffer[nivel] = 'a' + i;
            listarPalavras(no->filhos[i], buffer, nivel + 1);
        }
    }
}

void normalizar(const char* entrada, char* saida) {
    int j = 0;
    for (int i = 0; entrada[i] != '\0'; i++) {
        char c = entrada[i];
        if (c >= 'A' && c <= 'Z') c += 32; // Converte para minúsculo
        if (c >= 'a' && c <= 'z') saida[j++] = c; // Mantém apenas letras
    }
    saida[j] = '\0';
}

```

Por que é necessário normalizar?

Nossa Trie foi projetada para funcionar com letras minúsculas de 'a' a 'z', ou seja, apenas 26 caracteres. Assim, ao tentar usar palavras com **letras maiúsculas, espaços e acentos**, podem ocorrer erros de acesso fora dos limites, retornando possíveis falhas no funcionamento do programa.

A normalização corrige isso, pois, antes de inserir a palavra na Trie, é feita uma conversão para letras minúsculas, remoção dos espaços, eliminar ou substituir caracteres com acento e garantir que cada caractere esteja entre 'a' e 'z'.

Exemplo completo com função main

Para consolidar os conceitos apresentados até aqui, veja a seguir um exemplo completo em C, que demonstra a criação da estrutura, inserção de palavras normalizadas em uma árvore Trie e a listagem final dos elementos armazenados:

```

c

int main() {
    struct NoTrie* raiz = criarNo();
    char normalizada[100];

    normalizar("Pegadas de Lama", normalizada);
    inserir(raiz, normalizada);

    normalizar("Chave perdida", normalizada);
    inserir(raiz, normalizada);

    normalizar("Livro com página faltando", normalizada);
    inserir(raiz, normalizada);

    normalizar("Lençol manchado", normalizada);
    inserir(raiz, normalizada);

    normalizar("Gaveta perdida", normalizada);
    inserir(raiz, normalizada);

    char buffer[100]; // Tamanho máximo de palavra suportado
    listarPalavras(raiz, buffer, 0);

    return 0;
}

```

Saída esperada:

```

pegadasdelama
chaveperdida
livrocompaginafaltando
lencolmanchado
gavetaperdida

```

Para finalizar, a estrutura Trie é muito eficiente para cenários que envolvem grandes volumes de palavras, em especial, quando há muitos prefixos repetidos. Ela oferece um tempo de busca proporcional ao tamanho da palavra, independentemente da quantidade total de palavras armazenadas. Por isso, ela é muito usada em sistemas de autocompletar, filtros de spam e até em inteligências artificiais.

Hora de codar

Neste vídeo prático, você aplicará BST e Trie em C, construindo cada estrutura com exemplos de strings e testando inserções, buscas e percursos no terminal. Essa prática favorece a compreensão de suas diferenças e prepara você para o desafio de gerenciar pistas em Detective Quest.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Você irá implementar duas estruturas fundamentais de dados em C:

- BST: árvore binária de busca.
- Trie: árvore de prefixos para strings.

Para implementar as estruturas, você deve:

1. Definir as structs correspondentes a cada tipo de árvore, com os campos apropriados.
2. Implementar funções específicas para cada árvore, incluindo:
 1. Inserção de elementos.
 2. Busca por elementos.
3. Tipos de percurso (em ordem, pré-ordem, pós-ordem, conforme aplicável)
4. Testar as implementações no terminal, utilizando exemplos com strings variadas para observar o comportamento de cada estrutura.

Desafio: nível aventureiro

Neste vídeo-desafio, implemente o sistema de coleta de pistas em Detective Quest, integrando a árvore binária do mapa com uma BST de pistas. Ao explorar salas pré-definidas, seu código insere cada pista automaticamente e exibe todas, em ordem alfabética, ao final da jornada.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O que você vai fazer?

A Enigma Studios, especialista no desenvolvimento de jogos educativos de lógica e programação, está expandindo o título **Detective Quest** com um novo recurso: **coleta de pistas**.

Você, como desenvolvedor(a) técnico(a), será responsável por implementar o sistema de pistas coletadas durante a exploração da mansão. Para isso, ampliará o sistema anterior de árvore binária adicionando:

- Pistas associadas a cada cômodo da mansão.
- Uma árvore BST para armazenar e organizar as pistas conforme forem encontradas.

O objetivo do programa é permitir que o detetive explore a mansão, colete pistas espalhadas pelos cômodos e visualize ao final todos os indícios organizados de acordo com o alfabeto.

Requisitos funcionais

Seu programa em C deverá:

- Criar uma **árvore binária** que represente o mapa da mansão, com pistas já associadas aos cômodos.
- Criar uma **árvore BST** na qual serão armazenadas as pistas coletadas.
- Permitir a **exploração da mansão** a partir do "Hall de Entrada", escolhendo o próximo caminho entre esquerda (e), direita (d) ou sair (s).
- Adicionar automaticamente à árvore de pistas cada pista encontrada durante a jornada.
- Exibir **todas as pistas coletadas** em ordem alfabética ao final da exploração.

Cada cômodo possui:

- nome: string com o nome da sala (ex: "Sala de Estar", "Cozinha").
- pista: string opcional com o conteúdo da pista encontrada naquele cômodo.

Requisitos não funcionais

- **Usabilidade:** o programa deve apresentar mensagens claras sobre localização atual, pistas e opções disponíveis.
- **Legibilidade:** o código-fonte deve ser organizado, indentado e com nomes de variáveis compreensíveis.
- **Documentação:** comentário de cada parte importante do código, incluindo:
 - criarSala() – cria dinamicamente um cômodo com ou sem pista.
 - inserirPista() – insere uma nova pista na árvore de pistas.
 - explorarSalasComPistas() – controla a navegação entre salas e coleta de pistas.
 - exibirPistas() – imprime a árvore de pistas em ordem alfabética.

Simplificações para o nível aventureiro

- O mapa da mansão é fixo e pré-definido no main(). Ou seja, não há inserção ou remoção dinâmica de salas.
- Não há detecção automática do fim da jornada: o usuário deve optar por sair (s) a qualquer momento.
- As árvores não precisam ser balanceadas ou otimizadas.

Conceitos trabalhados

- **Árvore binária:** é a estrutura hierárquica para modelar os cômodos da mansão.
- **Árvore binária de busca (BST):** armazena as pistas de forma ordenada.

- **Structs:** usa tipos personalizados (Sala, PistaNode) para modelar os dados.
- **Alocação dinâmica:** usa malloc() para criação de elementos das árvores.
- **Recursividade:** exploração das árvores e exibição ordenada das pistas.
- **Modularização:** separação clara de responsabilidades por função.

Entregando seu projeto

1. **Desenvolva seu projeto no GitHub:** use o mesmo repositório do GitHub dos níveis anteriores.
2. **Atualize o arquivo do seu código:** edite o arquivo principal do seu projeto, inserindo o código completo já com as novas funcionalidades.
3. **Compile e teste:** faça isso com rigor, garantindo que todas as comparações e cálculos estejam corretos.
4. **Faça commit e push:** faça commit das suas alterações e envie (push) para o seu repositório no GitHub.
5. **Envie o link do repositório no GitHub:** faça isso através da plataforma SAVA.

Tutorial git

Você está prestes a aplicar os conceitos aprendidos para resolver um desafio prático no ambiente do GitHub. Veja as instruções gerais a seguir para acessar, aceitar e executar o desafio, de modo que sua solução esteja bem estruturada e documentada.

Dê o primeiro passo

Acesse o GitHub Classroom. Nesse ambiente, você terá acesso ao repositório padrão do desafio. Caso ainda não tenha uma conta no GitHub, não se preocupe: você pode criar uma grátis, clicando no [link](#).

Aceite o desafio

Tenha acesso ao repositório no GitHub, no qual você encontrará o repositório criado para o desenvolvimento do seu desafio.

Acesse o repositório

Clique no link do repositório para abrir o ambiente GitHub com a descrição do desafio e a estrutura modelo de arquivos e pastas que deve ser utilizada. Lembre-se: é esse o link que você deve enviar no SAVA.

Explore a estrutura do ambiente

Veja a estrutura organizada de pastas e arquivos necessários para o desenvolvimento do desafio.

Desenvolva o desafio

Utilize o GitHub CodeSpace para editar o arquivo do código-fonte e desenvolver o desafio. Certifique-se de que o código esteja organizado e funcional para resolver o problema proposto.

Entregue o desafio

Forneça o repositório do GitHub com todos os arquivos de código-fonte e conteúdos relacionados ao projeto. Certifique-se de que o repositório esteja bem estruturado, com pastas e arquivos nomeados de maneira clara e coerente. Envie o link para o repositório do seu desafio no GitHub.

Finalizando nosso estudo, aqui está a última orientação: comente todos os arquivos de código-fonte, pois isso demonstra o quanto você sabe sobre o funcionamento do código e facilita a correção por terceiros. Seus comentários devem explicar a finalidade das principais seções do código, o funcionamento de algoritmos complexos e o propósito de variáveis e funções utilizadas.

Assista agora ao último vídeo, com dicas e detalhes sobre a entrega do seu projeto. Até a próxima!



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Introdução a tabelas hash

Neste vídeo, apresentamos os conceitos fundamentais de tabelas hash, explicando chave-valor, função hash. Com analogias e aplicações práticas, você compreenderá a relevância dessa estrutura.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Conhecendo as tabelas hash

Na ciência da computação, tabelas hash são estruturas de dados fundamentais, conhecidas por oferecerem acesso muito rápido a informações. Para compreender sua importância, imagine a diferença entre procurar um número de telefone em uma lista telefônica tradicional, que está em ordem alfabética, e procurá-lo em uma agenda em que cada nome está associado a uma letra específica do alfabeto e você pode ir direto ao ponto certo. A segunda opção é uma analogia aproximada de como a tabela hash permite que acessemos um dado sem precisar percorrer toda a estrutura.

O que é uma tabela hash?

Também chamada de hash table, trata-se de uma estrutura que armazena dados de forma associativa, ou seja, cada valor é armazenado com base em uma **chave única**. Essa é processada por uma **função hash**, que transforma a chave (como um nome, número ou palavra) em um índice numérico, que determina a posição exata em que o valor será armazenado dentro da tabela.

Diferentemente de uma **lista** — em que para encontrar um elemento pode ser necessário percorrer item por item — ou de uma **árvore**, na qual é preciso seguir caminhos entre nós, a tabela hash permite localizar um elemento com muita rapidez, mesmo em grandes volumes de dados.

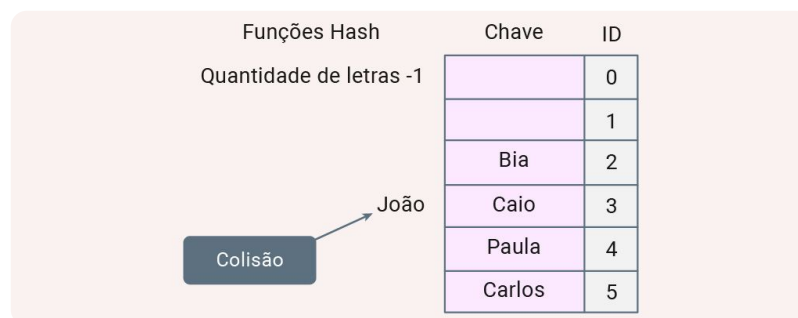


Tabela hash.

A imagem mostra uma tabela hash em que nomes são associados a índices com base na quantidade de letras - 1. Por exemplo, "Bia" vai para o índice 2, "Caio" para o 3, e "João" também para o 3, gerando colisão. Portanto, ela ilustra bem o funcionamento básico da função hash e a ocorrência de colisões.

Por que usar tabelas hash?

A principal motivação para o uso de tabelas hash é a **eficiência na busca por chave**. Em estruturas lineares como listas, o tempo de acesso médio cresce conforme o número de elementos aumenta. Já nas árvores, apesar de mais eficientes que listas, o tempo de acesso ainda depende da profundidade da estrutura.

As tabelas hash, por outro lado, podem localizar informações em **tempo constante** (denotado como $O(1)$ na notação de complexidade). Ou seja, o tempo de acesso não depende da quantidade de elementos armazenados. Isso é possível porque a chave é convertida em um índice que aponta para o local do dado.

Aplicações práticas

Mesmo que muitas vezes passem despercebidas, as tabelas hash estão presentes em diversas aplicações cotidianas. Confira!

1

Buscas em dicionários ou enciclopédias digitais

Ao digitar uma palavra, o sistema utiliza a chave (a palavra) para encontrar de maneira instantânea sua definição ou conteúdo relacionado.

2

Armazenamento e verificação de senhas

Em sistemas de login, senhas são, com frequência, transformadas por funções hash e armazenadas de forma segura. Quando o usuário digita a senha, o sistema a transforma, mais uma vez, e compara com o valor armazenado.

3

Sistemas de cache

Navegadores e aplicativos usam tabelas hash para armazenar com rapidez dados acessados com frequência.

4

Catálogos e inventários

Sistemas de vendas e bibliotecas usam chaves como códigos de produto ou para localizar informações com precisão e agilidade.

5

Verificação de duplicatas

Em uploads de arquivos, pode-se usar uma função hash para verificar se um arquivo já foi enviado, comparando os valores hash.

Em resumo, tabelas hash são estruturas que permitem o armazenamento e acesso a dados de forma rápida e organizada, com base em chaves únicas. Elas se destacam quando comparadas a listas e árvores pelo seu desempenho superior em buscas. Por essa razão, são muito utilizadas em sistemas que exigem rapidez e precisão, tornando-se uma ferramenta essencial no repertório de qualquer desenvolvedor.

Dominar esse conceito é dar um passo importante rumo à construção de algoritmos mais eficientes e inteligentes. Entender tabelas hash é como aprender a usar um atalho poderoso no mundo da computação.

Associação chave-valor e funções hash

Neste vídeo, você vai explorar implementações de funções hash em C, aprendendo hash simples e ponderado para strings. Será detalhado como transformar chaves em índices, propriedades de boas funções hash e seu impacto em colisões, desempenho e eficiência de sua tabela.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Introdução à chave-valor

Em muitas aplicações da ciência da computação, precisamos armazenar e acessar informações de maneira eficiente. A estrutura **chave → valor** é uma solução comum para esse problema. Nela, os dados são organizados de modo que cada valor esteja associado a uma **chave única**, que serve como identificador. Esse modelo é usado em tabelas hash, dicionários, mapas e outras estruturas semelhantes.



Exemplo

Imagine um catálogo em que a chave seja o nome de um livro e o valor contenha suas informações (autor, ano e editora). Em vez de procurar manualmente pelo título em uma lista, podemos usar a chave para localizar o valor correspondente de forma muito mais rápida.

O papel da função hash

Primeiro, vamos entender que a **função hash** é o mecanismo que torna a eficiência da chave-valor possível. Ela é responsável por **converter uma chave (em geral, uma string, como "Suspeito1" ou "Suspeito2") em um número inteiro**, que indicará a posição (índice) em que o valor será armazenado ou buscado dentro de um vetor interno.

Esse processo é chamado de **mapeamento**, e é indispensável que seja feito de maneira eficiente para o bom desempenho da tabela hash. A chave, portanto, não é usada de forma direta, mas transformada por essa função antes de acessar o armazenamento.

Propriedades de uma boa função hash

São elas:

Determinismo

A função deve sempre gerar o mesmo valor de hash para a mesma chave. Logo, se "João" for convertido em 35 uma vez, sempre deverá resultar 35.

Eficiência

O cálculo da função hash deve ser rápido, já que ela será usada com frequência durante operações de inserção, busca e remoção.

Dispersão uniforme

As chaves devem ser distribuídas de forma uniforme ao longo do vetor. Assim, se muitas chaves forem mapeadas para o mesmo índice (colisões), o desempenho da tabela pode se degradar logo.

Implementando funções hash simples em C

Acompanhe alguns exemplos simples de funções hash para strings, usando a soma dos valores ASCII dos caracteres da string:

```
c
// Função hash básica: soma dos caracteres e uso de módulo
int hash_simples(const char* str, int tamanho_tabela) {
    int soma = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        soma += str[i];
    }
    return soma % tamanho_tabela;
}
```

No exemplo, a função percorre cada caractere da string e acumula seu valor numérico (com base na tabela ASCII). O resultado é, então, reduzido ao intervalo da tabela (por exemplo, 0 a 99 para uma tabela de 100 posições), usando o operador **módulo** (**% tamanho_tabela**).

Podemos também aplicar pesos para dar mais importância à posição dos caracteres:

```
c
int hash_ponderado(const char* str, int tamanho_tabela) {
    int hash = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        hash += str[i] * (i + 1);
    }
    return hash % tamanho_tabela;
}
```

Nesse caso, o valor de cada caractere é multiplicado por sua posição na string, o que tende a gerar uma distribuição um pouco melhor para certas combinações.

Influência da função hash na eficiência

A escolha da função hash tem impacto direto na **eficiência geral da tabela hash**. Uma função ruim, que gera muitos índices repetidos, resulta **colisões**, que forçam o uso de técnicas adicionais, como listas encadeadas, sondagem linear ou dupla. Isso afeta a velocidade das operações e, em casos extremos, pode tornar a tabela hash tão lenta quanto uma lista comum.

Por outro lado, uma função hash bem planejada e adaptada ao tipo de dados armazenados proporciona acesso rápido, até mesmo em grandes volumes de informação. Por isso, projetar boas funções hash é uma etapa crítica na construção de sistemas de alta performance, desde bancos de dados até caches de páginas da web e verificadores de integridade de arquivos.



Comentário

Vale destacar que o estudo das funções hash vai muito além de um simples exercício técnico. Ele desempenha um papel fundamental na criação de soluções eficientes para problemas concretos de organização e busca de dados.

Tabela hash com encadeamento (chaining)

Neste vídeo, você verá como implementar uma tabela hash em C com encadeamento, definindo structs de lista ligada, função hash e operações de inserir, buscar e remover. A técnica resolve colisões por listas em cada índice, oferecendo flexibilidade e desempenho para relacionar dados.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Introdução a tabelas hash com encadeamento

Quando lidamos com **grandes volumes de dados** e queremos fazer buscas rápidas, as **tabelas hash** surgem como uma das estruturas mais eficientes. Elas permitem que uma chave — como um nome ou número — seja convertida em um **índice de um vetor** e usada para localizar um valor com **tempo médio de acesso constante ($O(1)$)**.

No entanto, essa eficiência depende do bom tratamento de um problema inevitável: as **colisões**.

O que são colisões?

Quando **duas ou mais chaves diferentes** são transformadas, pela **função hash**, no **mesmo índice** da tabela ocorre uma colisão. Como o vetor só pode armazenar um elemento por posição, precisamos de uma maneira de **armazenar múltiplos dados na mesma posição**.

Encadeamento

Resolve colisões utilizando, em cada posição do vetor, uma **lista ligada**. Assim, quando várias chaves forem direcionadas ao mesmo índice, elas são adicionadas como nós dessa lista.

Vamos entender melhor! Imagine uma caixa de correio com várias cartas dentro. A "caixa" é a posição do vetor, e cada "carta" é um nó da lista com uma chave.

Essa solução é flexível, simples de implementar e eficiente na prática.

Implementação

Vamos implementar uma tabela hash simples que armazena **nomes (strings)**. Para isso, começamos definindo nossas estruturas:

```
c
#include
#include
#include

#define TAMANHO_TABELA 10

// Estrutura de um nó da lista ligada
typedef struct Nodo {
    char nome[50];           // Armazena o nome
    struct Nodo* proximo;    // Ponteiro para o próximo nó
} Nodo;

// A tabela hash é um vetor de ponteiros para Nodo
Nodo* tabela_hash[TAMANHO_TABELA];
```

Aqui temos:

- Um vetor de ponteiros (tabela_hash) com 10 posições.
- Cada posição pode apontar para o **início de uma lista ligada**.
- Cada Nodo da lista guarda um nome e aponta para o próximo nome (se houver).

Função hash

Precisamos transformar uma string (nome) em um índice de vetor. Para isso, criamos uma **função hash** simples, que soma os valores dos caracteres e aplica o operador % para restringir ao tamanho da tabela. Veja!

```
c
int funcao_hash(const char* chave) {
    int soma = 0;
    for (int i = 0; chave[i] != '\0'; i++) {
        soma += chave[i];
    }
    return soma % TAMANHO_TABELA;
}
```

Agora que vimos a função que transforma as strings em um índice de vetor, vejamos na prática como isso se dá!



Exemplo

Se o nome for "Ana", a soma dos valores ASCII será algo como $65 + 110 + 97 = 272$. Com $TAMANHO_TABELA = 10$, temos $272 \% 10 = 2$. Ou seja, "Ana" será armazenada na posição 2 da tabela.

Inserir um nome

Para inserir um nome, criamos um nó e o adicionamos ao **início da lista ligada** correspondente à posição gerada pela função hash:

```
c
void inserir(const char* nome) {
    int indice = funcao_hash(nome); // Descobre onde armazenar
    // Cria um nó
    Nodo* novo = (Nodo*)malloc(sizeof(Nodo));
    strcpy(novo->nome, nome);

    // Insere no início da lista (head)
    novo->proximo = tabela_hash[indice];
    tabela_hash[indice] = novo;
}
```

No código que vimos:

- Não estamos sobrescrevendo o valor já presente no índice, mas, sim, **encadeando** os dados.
- Inserir no início da lista é rápido (afinal, não é necessário percorrer a lista toda).
- É possível que nomes diferentes com o mesmo índice sejam organizados como uma pequena fila, como consequência.

Buscar um nome

A procura percorre a lista da posição indicada pela função hash e compara nome por nome até encontrar o desejado. Confira!

```
c
Nodo* buscar(const char* nome) {
    int indice = funcao_hash(nome);
    Nodo* atual = tabela_hash[indice];

    while (atual != NULL) {
        if (strcmp(atual->nome, nome) == 0) {
            return atual; // Encontrou
        }
        atual = atual->proximo;
    }

    return NULL; // Não está na lista
}
```

Explicando:

- Usamos strcmp para comparar strings.
- Se o nome for encontrado, retornamos um ponteiro para o nó.
- Se não, retornamos NULL, indicando que o nome não foi cadastrado.

Remover um nome

Excluir um item de uma lista ligada exige um pouco mais de cuidado. Assim, precisamos manter a referência do elemento anterior para reencadear a lista após a remoção. Acompanhe!

```
c
void remover(const char* nome) {
    int indice = funcao_hash(nome);
    Nodo* atual = tabela_hash[indice];
    Nodo* anterior = NULL;

    while (atual != NULL) {
        if (strcmp(atual->nome, nome) == 0) {
            if (anterior == NULL) {
                tabela_hash[indice] = atual->proximo; // Era o primeiro
            } else {
                anterior->proximo = atual->proximo; // "Pula" o nó atual
            }
            free(atual);
            printf("'%s' removido.\n", nome);
            return;
        }
        anterior = atual;
        atual = atual->proximo;
    }

    printf("'%s' não encontrado.\n", nome);
}
```

Explicando passo a passo:

- A função procura o nome desejado.
- Se for o primeiro da lista, apenas o ponteiro da tabela é atualizado.
- Se estiver no meio ou final, o nome anterior aponta para o próximo do nome atual.
- A memória com free é liberada.

Função main – cadastro de nomes

Agora, vamos usar essas funções em um miniprograma que cadastra e manipula nomes:

```
c
int main() {
    inserir("Ana");
    inserir("Bruno");
    inserir("Carlos");
    inserir("Amanda"); // Pode colidir com "Ana"

    printf("Buscando 'Bruno': %s\n", buscar("Bruno") ? "Encontrado" : "Não encontrado");
    printf("Buscando 'João': %s\n", buscar("João") ? "Encontrado" : "Não encontrado");

    remover("Carlos");
    remover("João");

    return 0;
}
```

O que acontece aqui:

- "Ana" e "Amanda" podem acabar no mesmo índice — e o encadeamento resolve isso.
- "Carlos" será removido com sucesso.
- Ao tentar remover "João", a função nos informa que ele não foi encontrado.

A **tabela hash com encadeamento** é uma estrutura robusta para situações em que precisamos armazenar dados associados a uma chave, com **acesso e manipulação rápidos**. Sua implementação com **listas ligadas** torna o sistema flexível e resistente a colisões, que são comuns em qualquer função hash.

Tal modelo é usado em **cadastros, dicionários, catálogos, banco de dados e sistemas de login**, entre muitos outros contextos reais. Ao entender sua lógica, você amplia sua capacidade de desenvolver **sistemas eficientes e escaláveis**.

Tabela hash com endereçamento aberto

Este vídeo demonstra a tabela hash com endereçamento aberto e sondagem linear em C. Você verá structs de entrada, inserção, busca e remoção em vetor interno, e aprenderá a manejar estados livre, ocupado e removido, garantindo integridade e agilidade no acesso.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Introdução a tabelas de encadeamento aberto

Uma **tabela hash** é uma estrutura de dados que permite associar **chaves a valores** de forma muito eficiente, com operações de inserção, busca e remoção com **tempo médio constante ($O(1)$)**.

Mas como garantir eficiência quando ocorrem **colisões**? Uma abordagem alternativa ao encadeamento (listas ligadas) é o **endereçamento aberto**, que mantém **todos os dados dentro do próprio vetor da tabela**, sem usar listas auxiliares.

O que é endereçamento aberto?

Quando duas chaves produzem o mesmo índice (colisão), em vez de criar listas, a estrutura procura outra posição livre no vetor seguindo um padrão de busca (linear, quadrática, duplo hashing etc.). Usaremos aqui a sondagem linear (ou linear probing): se a posição i estiver ocupada, ela tenta $i+1$, depois $i+2$, e assim por diante, até encontrar uma posição livre.

Estrutura de dados

Vamos armazenar nomes em uma tabela hash simples. Primeiro, criamos uma struct para representar uma entrada da tabela. Acompanhe!

```
c
#include
#include
#include

#define TAMANHO_TABELA 10
typedef struct {
    char nome[50];
    int ocupado; // 0 = vazio, 1 = ocupado, -1 = removido
} Entrada;

Entrada tabela_hash[TAMANHO_TABELA];
```

Explicando:

- Cada posição do vetor contém uma Entrada, que armazena um nome e um indicador de estado (ocupado).
- Usamos -1 para sinalizar que ali já houve algo, mas foi removido. Isso é importante para manter a busca funcionando de maneira correta.

Função hash

Usamos a mesma função simples para transformar a chave (nome) em um índice do vetor:

```
c
int funcao_hash(const char* chave) {
    int soma = 0;
    for (int i = 0; chave[i] != '\0'; i++) {
        soma += chave[i];
    }
    return soma % TAMANHO_TABELA;
}
```

Inserção com sondagem linear

Agora vamos à função de inserir nomes. Se a posição estiver ocupada, seguimos procurando a próxima vaga. Veja!

```
c
void inserir(const char* nome) {
    int indice = funcao_hash(nome);
    for (int i = 0; i < TAMANHO_TABELA; i++) {
        int pos = (indice + i) % TAMANHO_TABELA;

        if (tabela_hash[pos].ocupado == 0 || tabela_hash[pos].ocupado == -1) {
            strcpy(tabela_hash[pos].nome, nome);
            tabela_hash[pos].ocupado = 1;
            return;
        }
    }
    printf("Tabela cheia. Não foi possível inserir '%s'.\n", nome);
}
```

Explicação detalhada:

- Começamos do índice gerado pela função hash.
- Se a posição estiver livre (0) ou marcada como removida (-1), inserimos um nome.
- Tentamos a próxima posição ($pos = (índice + i) \% tamanho$), caso a posição esteja ocupada.
- Se percorrermos toda a tabela sem sucesso, significa que ela está cheia.

Buscar um nome

A procura também segue o mesmo padrão de sondagem linear:

```
c
int buscar(const char* nome) {
    int indice = funcao_hash(nome);

    for (int i = 0; i < TAMANHO_TABELA; i++) {
        int pos = (indice + i) % TAMANHO_TABELA;

        if (tabela_hash[pos].ocupado == 0) {
            return -1; // Paramos: posição nunca foi usada
        }

        if (tabela_hash[pos].ocupado == 1 && strcmp(tabela_hash[pos].nome, nome) ==
0) {
            return pos;
        }
    }

    return -1; // Não encontrou
}
```

Aqui cabe um ponto ao qual devemos nos atentar!



Atenção

Se encontramos uma posição vazia (0), podemos parar o dado, pois não está na tabela. Se achamos uma posição com o nome certo, retornamos seu índice. Se o campo está removido (-1) ou ocupado com outro valor, continuamos procurando.

Remover um nome

Para retirar, procuramos pelo nome e marcamos a posição como removida (-1):

```
c
void remover(const char* nome) {
    int pos = buscar(nome);
    if (pos == -1) {
        printf("'%' não encontrado.\n", nome);
        return;
    }

    tabela_hash[pos].ocupado = -1;
    printf("'%' removido da tabela.\n", nome);
}
```

Por que marcar como removido?

- Se colocarmos como 0 (vazio), a busca futura pode terminar cedo demais, achando que o item nunca esteve ali.

- Usar -1 permite manter o ciclo de busca funcionando como o esperado.

Função main

A seguir, temos um exemplo prático mais completo em C, que ilustra o uso de uma tabela hash com operações de inserção, busca e remoção. Esse trecho permite observar como a estrutura se comporta em situações reais, incluindo casos de colisão e tentativas de remoção de elementos inexistentes. Vamos lá!

```
c

int main() {
    // Inicializa a tabela
    for (int i = 0; i < TAMANHO_TABELA; i++) {
        tabela_hash[i].ocupado = 0;
    }

    inserir("Ana");
    inserir("Bruno");
    inserir("Carlos");
    inserir("Amanda"); // Pode colidir com Ana

    printf("Buscando 'Bruno': %s\n", buscar("Bruno") != -1 ? "Encontrado" : "Não encontrado");
    printf("Buscando 'João': %s\n", buscar("João") != -1 ? "Encontrado" : "Não encontrado");

    remover("Carlos");
    remover("João");

    return 0;
}
```

A **tabela hash com endereçamento aberto** é uma alternativa elegante ao encadeamento: todas as informações são mantidas dentro do próprio vetor, sem uso de ponteiros ou listas. Isso simplifica o gerenciamento da memória e melhora o cache do processador.

Por outro lado, essa abordagem exige controle cuidadoso de colisões e atenção especial ao estado das posições (livre, ocupado e removido).

Essa técnica é bastante usada em bancos de dados simples, compiladores, mecanismos de busca e em qualquer sistema que precise associar com rapidez uma **chave a um valor como logins, dicionários e catálogos, entre outros**.

Hora de codar

Neste vídeo prático, será criada uma tabela hash simples em C, definindo funções hash, tratamento de colisões por encadeamento e sondagem, e testando inserção, busca e remoção no terminal. Essa aplicação aprofunda o modelo chave-valor e prepara para o desafio mestre de associação de pistas.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Você irá implementar uma tabela hash em C, explorando duas estratégias de tratamento de colisões:

- Encadeamento (chaining).
- Sondagem linear (linear probing).

Essa implementação terá como foco a manipulação de dados do tipo string.

Para implementar a tabela em hash você deve:

1. Definir uma função hash adequada para distribuir as strings de modo uniforme.
2. Criar funções de inserção, busca e remoção, adaptadas tanto para encadeamento como para sondagem linear.
3. Testar a tabela hash no terminal, utilizando exemplos de palavras para observar o comportamento das diferentes abordagens de colisão.

Desafio: nível mestre

Neste vídeo de desafio mestre, você integrará árvore binária do mapa, BST de pistas e tabela hash de suspeitos, conduzindo a exploração interativa da mansão, coleta automática de pistas e acusação final. Seu programa confirmará evidências e encerrará a investigação em C.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O que você vai fazer?

A Enigma Studios está finalizando o capítulo mais desafiador de **Detective Quest**, e escalaram você para um upgrade final: permitir que o jogador **colecione pistas, relacione-as a suspeitos** e tome uma decisão lógica sobre **quem é o culpado**.

Sua missão como desenvolvedor(a) técnico(a) é programar o sistema completo que:

- Permite explorar uma **mansão representada como uma árvore binária**.
- Armazena as pistas encontradas em uma **árvore binária de busca**.
- Associa cada pista a um suspeito usando uma **tabela hash**.

Ao final da exploração, o jogador deve indicar quem é o culpado, e o sistema avaliará, de maneira automática, **se há pistas suficientes para sustentar a acusação**.

Requisitos funcionais

Seu programa em C deverá:

- Criar uma **árvore binária de cômodos**, cada um com um nome exclusivo.
- Ao visitar uma sala, identificar e exibir uma **pista específica associada à sala** (se existir).
- Armazenar cada pista coletada em uma **árvore BST** de forma ordenada.
- Criar uma **tabela hash** com as pistas, como chaves, e seus respectivos **suspeitos** como valores.
- Permitir ao jogador **explorar os cômodos de modo interativo**, escolhendo ir para a esquerda (e), direita (d) ou sair (s).
- Ao final, listar as pistas coletadas e solicitar ao jogador a **acusação de um suspeito**.
- Verificar se, pelo menos, **duas pistas apontam para o suspeito acusado** e exibir a mensagem correspondente ao desfecho.

Cada cômodo possui:

- **nome** identificador do cômodo.
- Uma pista estática associada a esse nome, definida por lógica no código.

Requisitos não funcionais

- **Usabilidade:** o jogo deve mostrar mensagens informativas sobre cada passo da exploração e as ações finais do julgamento.
- **Legibilidade:** o código deve ser bem indentado, com estruturas modulares e nomes representativos.
- **Documentação:** comentários explicativos devem existir nas seguintes funções:
 - criarSala() – cria dinamicamente um cômodo.
 - explorarSalas() – navega pela árvore e ativa o sistema de pistas.
 - inserirPista() / adicionarPista() – insere a pista coletada na árvore de pistas.
 - inserirNaHash() – insere associação pista/suspeito na tabela hash.
 - encontrarSuspeito() – consulta o suspeito correspondente a uma pista.
 - verificarSuspeitoFinal() – conduz à fase de julgamento final.

Simplificações para o nível mestre

- O mapa da mansão é fixo e é montado de forma manual no main().
- Não há inserção ou remoção dinâmica de salas nem de suspeitos.

- As pistas associadas a salas são definidas por regras codificadas (sem entrada manual).
- A exploração termina quando o jogador escolhe sair (s).

Conceitos trabalhados

- **Árvore binária:** representa a estrutura de navegação da mansão.
- **Árvore de busca binária (BST):** armazena as pistas coletadas em ordem.
- **Tabela hash:** associa pistas a suspeitos de forma eficiente.
- **Structs e ponteiros:** utiliza alocação dinâmica e manipulação de dados compostos.
- **Recursividade:** Exploração da árvore e contagem de pistas por suspeito.
- **Condicionais e loops:** Controle da lógica de jogo e verificação de hipóteses.

Entregando seu projeto

1. **Desenvolva seu projeto no GitHub:** use o mesmo repositório do GitHub dos níveis anteriores.
2. **Atualize o arquivo do seu código:** edite o arquivo principal do seu projeto, inserindo o código completo já com as novas funcionalidades.
3. **Compile e teste:** faça isso com rigor, garantindo que todas as comparações e cálculos estejam corretos.
4. **Faça commit e push:** faça commit das suas alterações e envie (push) para o seu repositório no GitHub.
5. **Envie o link do repositório no GitHub:** faça isso através da plataforma SAVA.

Tutorial git

Você está prestes a aplicar os conceitos aprendidos para resolver um desafio prático no ambiente do GitHub. Veja as instruções gerais a seguir para acessar, aceitar e executar o desafio, de modo que sua solução esteja bem estruturada e documentada.

Dê o primeiro passo

Acesse o GitHub Classroom. Nesse ambiente, você terá acesso ao repositório padrão do desafio.

Caso ainda não tenha uma conta no GitHub, não se preocupe: você pode criar uma grátis, clicando no [link](#).

Aceite o desafio

Tenha acesso ao repositório no GitHub, no qual você encontrará o repositório criado para o desenvolvimento do seu desafio.

Acesse o repositório

Clique no link do repositório para abrir o ambiente GitHub com a descrição do desafio e a estrutura modelo de arquivos e pastas que deve ser utilizada. Lembre-se: é esse o link que você deve enviar no SAVA.

Explore a estrutura do ambiente

Veja a estrutura organizada de pastas e arquivos necessários para o desenvolvimento do desafio.

Desenvolva o desafio

Utilize o GitHub CodeSpace para editar o arquivo do código-fonte e desenvolver o desafio. Certifique-se de que o código esteja organizado e funcional para resolver o problema proposto.

Entregue o desafio

Forneça o repositório do GitHub com todos os arquivos de código-fonte e conteúdos relacionados ao projeto. Certifique-se de que o repositório esteja bem estruturado, com pastas e arquivos nomeados de maneira clara e coerente. Envie o link para o repositório do seu desafio no GitHub.

Finalizando nosso estudo, aqui está a última orientação: comente todos os arquivos de código-fonte, pois isso demonstra o quanto você sabe sobre o funcionamento do código e facilita a correção por terceiros. Seus comentários devem explicar a finalidade das principais seções do código, o funcionamento de algoritmos complexos e o propósito de variáveis e funções utilizadas.

Assista agora ao último vídeo, com dicas e detalhes sobre a entrega do seu projeto. Até a próxima!



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Considerações finais

Parabéns pela conquista!

Parabéns por chegar ao fim dessa jornada em “investigação binária”! Você enfrentou níveis desafiadores, consolidando desde a implementação de árvores binárias simples até árvores de busca (BST), Tries e tabelas hash. Ao longo do caminho, você revisitou conceitos centrais de C (como structs, ponteiros, alocação dinâmica, recursão e modularização) e viu na prática como cada estrutura se encaixa no fluxo de um jogo de detetive: construir o mapa, coletar e ordenar pistas, e relacioná-las a suspeitos.

Esse percurso reforçou suas habilidades de programação, assim como aguçou sua capacidade de escolher a melhor estrutura de dados para cada problema. Você aprendeu a controlar altura e balanceamento de árvores, otimizar buscas por prefixo com Tries e lidar com colisões em tabelas hash. Mais do que códigos funcionando, você desenvolveu senso crítico sobre desempenho, legibilidade e organização de software.

Continue praticando! A repetição e a curiosidade são a chave para transformar teoria em habilidade sólida. Desejamos sucesso nos próximos desafios e siga desvendando mistérios — sejam eles em mansões virtuais ou na própria ciência da computação!

Entregando seu projeto

Para concluir o desafio, é necessário que você faça o commit no GitHub do que você desenvolveu. Certifique-se de organizar e documentar seu repositório da maneira certa. Essa etapa ajuda a demonstrar seu aprendizado.

Por fim, cadastre na Guia de trabalhos da SAVA o link do seu repositório.

Atenção! Caso você atualize o seu repositório, não é necessário alterá-lo na SAVA.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Continue praticando

- **Interpretador de expressões matemáticas:** crie uma árvore de expressão (AST) que leia, avalie e simplifique operações aritméticas.
- **Sistema de autocompletar com Trie:** desenvolva sugestões de palavras por frequência de uso, navegando prefixos de forma eficiente.
- **Simulação de sistema de arquivos:** implemente árvore de diretórios e arquivos com operações de navegação, criação e remoção.

Referências

- BACKES, A. **Estrutura de dados descomplicada – em Linguagem C**. Rio de Janeiro: Gen, 2016.

- CORMEN, T. H. **Algoritmos:** teoria e prática. 3. ed. Rio de Janeiro: Gen, 2012.
- FERRARI, R. *et al.* **Estrutura de dados com jogos.** Rio de Janeiro: Gen, 2014.