Listas e ordenação

Profª Daisy Albuquerque

1. Itens iniciais

Objetivos

- Compreender a implementação, na linguagem C, de listas lineares (estáticas e dinâmicas), listas encadeadas e as operações de inserção, remoção, percorrimento e busca sequencial de elementos.
- Comparar, utilizando a linguagem C, vetores e listas encadeadas por meio de operações de inserção e remoção e implementar a busca binária de forma iterativa e recursiva.
- Analisar, na linguagem C, diferentes métodos de ordenação, implementando-os para observar sua eficiência nos melhores, piores e médios casos com base no número de comparações e desempenho.
  $vetor[j] > vetor[j+1]$

Introdução

Olá! Boas-vindas ao desafio de programação, no qual você será responsável por desenvolver um modo de sobrevivência inspirado no universo do Free Fire utilizando a linguagem C!

A TechNova, empresa de inovação em jogos digitais, contratou você para criar a lógica que sustenta a sobrevivência dos jogadores em uma ilha repleta de perigos. Sua missão é construir, com eficiência e estratégia, os sistemas que vão desde a organização do inventário até a construção da torre de fuga.

Em tecnologia da informação, é indispensável saber manipular dados, organizar estruturas, otimizar algoritmos e tomar decisões baseadas em desempenho. Assim, neste desafio prático, você estará diante de situações reais que pessoas desenvolvedoras enfrentam diariamente ao criar jogos, sistemas interativos e aplicações otimizadas.

Iniciando nossos estudos, acompanhe, no vídeo a seguir, o desenvolvimento de um mini game em C inspirado em Free Fire, aplicando estruturas de dados, busca, ordenação e modularização em três níveis progressivos: organização, eficiência e estratégia para escapar da ilha.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

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

Cenário

O modo sobrevivência do Free Fire está prestes a ganhar uma nova versão, e a TechNova quer garantir que a experiência seja **realista e desafiadora**. Para isso, os desenvolvedores precisam de uma base sólida de lógica e estrutura de dados para representar, com fidelidade, os principais elementos da ilha: inventário, itens, recursos e estratégias de fuga.

A empresa selecionou você para implementar essa lógica. Portanto, prepare-se, pois, ao longo do desenvolvimento, você enfrentará desafios típicos de profissionais da área de programação: organizar dados em memória, comparar estruturas, otimizar buscas e ordenações, além de modularizar seu código de forma legível e escalável.

Sua missão

Você deverá desenvolver um mini game em C que simule a experiência de sobrevivência em uma ilha hostil. O projeto está dividido em três etapas progressivas que representam a evolução do jogador, desde a chegada à ilha até o momento decisivo da fuga. Cada fase trará novos aprendizados e exigirá o domínio de estruturas de dados, a análise de algoritmos e boas práticas de desenvolvimento.

O programa ainda deverá:

- Implementar a mochila de inventário do jogador, utilizando structs e listas sequenciais para armazenar alimentos, armas e ferramentas.
- Comparar listas encadeadas com vetores, implementando soluções mais flexíveis e eficientes. O programa também desenvolverá algoritmos de busca binária e ordenação simples.
- Aplicar algoritmos mais avançados, como Selection Sort, além de utilizar busca binária em dados ordenados.

Por último, lembre-se: o desafio envolve ordenar componentes do inventário por nome, tipo ou prioridade e construir logicamente a torre de fuga, integrando todos os elementos desenvolvidos.

O desafio vai começar. Boa sorte, sobrevivente da programação!

Listas lineares: estáticas e dinâmicas

Assista, no vídeo a seguir, às listas lineares estáticas e dinâmicas em C, explorando suas diferenças, seus usos e suas vantagens e como alocar, acessar e redimensionar elementos em memória utilizando vetores e ponteiros.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

De forma bastante simples, podemos dizer que uma **lista linear** é uma coleção de elementos armazenados de forma sequencial.

Na programação, uma das tarefas mais comuns é organizar dados de forma sequencial, ou seja, um elemento após o outro. Para isso, utilizamos listas lineares que armazenam, acessam e manipulam dados com facilidade.

Elas podem ser **estáticas** ou **dinâmicas**, dependendo da forma como utilizam a memória.

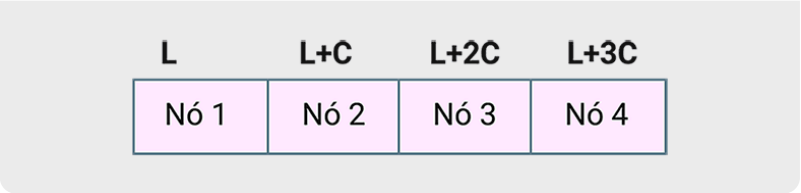
Lista linear estática

É uma estrutura de dados em que os elementos são armazenados de forma sequencial em posições contíguas da memória (isto é, ocupando lugares “vizinhos” na memória), com tamanho fixo determinado no momento da sua declaração.

Em linguagens como a C, listas linaeres estáticas são implementadas por meio de vetores (arrays), e seu principal benefício é o **acesso direto e rápido** aos elementos por índice. Confira o exemplo a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Lista linear estática.

Na imagem, se cada nó ocupa C posições de memória e se o primeiro nó está alocado na posição L, então os demais nós são alocados em sequência, e cada um deles começa exatamente C posições depois do anterior: L, L+C, L+2C e assim por diante.

No entanto, como não é possível aumentar ou diminuir o tamanho durante a execução do programa, seu uso é mais indicado em situações em que a quantidade de dados a ser armazenada já é conhecida.

Vamos pensar neste exemplo: imagine um armário com várias **prateleiras fixas**. Cada alimento ocupa uma posição específica, conforme a imagem a seguir. Você pode colocar um item em cada espaço: o arroz na primeira, o feijão na segunda, o macarrão na terceira e assim por diante. Depois que todas as posições estiverem ocupadas, você não conseguirá adicionar mais nada sem tirar algo antes.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

Armários com prateleiras fixas e mantimentos.

Nosso exemplo nos traz a ideia de uma lista linear estática, também chamada de **vetor (ou array) em C**. Ela é chamada de estática porque o tamanho é fixado no momento da declaração, e não pode ser mudado durante a execução do programa.

Exemplo prático em C

Vamos declarar um vetor de inteiros com 5 posições. Cada elemento pode ser acessado por seu índice, que começa em zero. Veja!

```c
int numeros[5] = {10, 20, 30, 40, 50};
```

Na linha, será impresso o número 30, que corresponde à terceira posição da lista, mas com o índice 2, pois o índice inicia em 0.

```c
printf("%d", numeros[2]);  // Imprime 30
```

Lista linear dinâmica

É a estrutura de dados que armazena elementos de forma sequencial, mas com capacidade de ajustar seu tamanho durante a execução do programa, alocando e desalocando memória conforme necessário.

Diferentemente das listas estáticas, ela oferece maior flexibilidade, sendo ideal para situações em que a quantidade de dados varia.

Em C, ela é implementada usando ponteiros e funções, como:

Malloc
Função usada para alocar dinamicamente um bloco de memória de tamanho especificado. Retorna um ponteiro para seu início.

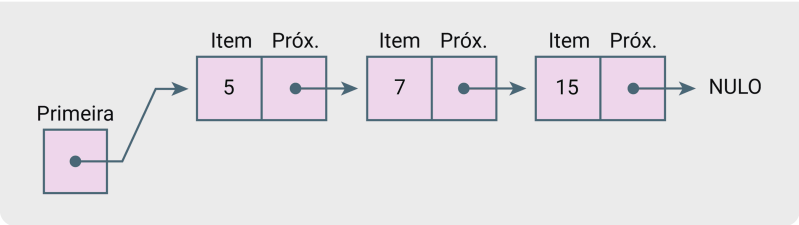
Realloc
Função usada para redimensionar um bloco de memória já alocado, preservando seu conteúdo até o menor dos tamanhos, podendo ser antigo e novo.

Free
Função usada para liberar um bloco de memória já alocado, tornando-o disponível para uso futuro.

Malloc, Realloc e Free permitem ao programador criar estruturas que crescem ou diminuem conforme os dados são inseridos ou removidos. Veja, agora, um exemplo de lista dinâmica.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Lista linear dinâmica.

Agora, acompanhe este exemplo: imagine que você tem uma **caixa dobrável** que pode expandir à medida que você adiciona mais itens. A caixa começa pequena, mas vai se aumentando de acordo com a demanda.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

Caixas dobráveis.

O exemplo aborda uma lista linear dinâmica, que aloca espaço na memória **conforme necessário** durante a execução do programa.

Exemplo prático em C

O objetivo é criar uma lista linear dinâmica de inteiros, que começa com 3 posições e, em seguida, é expandida para 5 posições com o uso de alocação dinâmica de memória.

Esse código constrói uma lista de inteiros de forma dinâmica e flexível. Primeiro, ele cria uma lista com 3 elementos e os preenche com os valores 10, 20 e 30. Depois, ela **aumenta a lista** para comportar 5 elementos e adiciona os valores 40 e 50. Vamos lá!

1. Criar a lista com 3 posições:

```c
int* numeros = (int*) malloc(3 * sizeof(int));
```

Vamos entender, passo a passo, o que está acontecendo nessa linha de código com a função malloc.

- A função malloc (memory allocation) está alocando, de maneira dinâmica, espaço para 3 inteiros na memória.
- sizeof(int) retorna o tamanho (em bytes) de um inteiro no sistema (em geral, 4 bytes).
- A multiplicação 3 * sizeof(int) garante espaço suficiente para 3 elementos.
- (int*) é um type cast que converte o ponteiro genérico retornado por malloc em um ponteiro para inteiro.

2. Atribuir os valores iniciais:

```
c

numeros[0] = 10;
numeros[1] = 20;
numeros[2] = 30;
```

Os comandos vistos armazenam os valores 10, 20 e 30 nas 3 posições alocadas.

3. Expandir a lista para 5 posições:

```
c

numeros = (int*) realloc(numeros, 5 * sizeof(int));
```

Veja agora como a função **realloc** atua ao redimensionar dinamicamente um bloco de memória.

Mas antes, vamos entender que realloc (realocação) é usada para redimensionar o bloco de memória apontado por números. Para isso,

- Solicitamos espaço para 5 inteiros.
- O conteúdo anterior (10, 20, 30) é preservado de modo automático (se houver memória suficiente), e duas novas posições são adicionadas.
- O novo endereço retornado (que pode ser diferente do original) é, mais uma vez, armazenado em números.

4. Atribuir novos valores:

```
c

numeros[3] = 40;
numeros[4] = 50;
```

Por fim, os novos espaços da lista são preenchidos com os valores 40 e 50.

Ao final, o vetor dinâmico números contém:

```
[10, 20, 30, 40, 50]
```

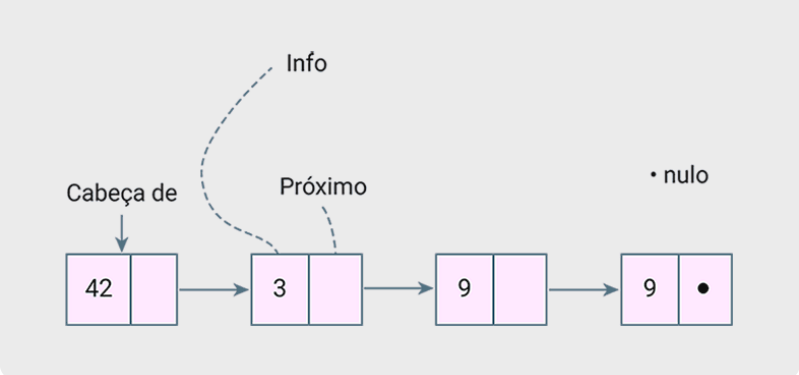
Listas encadeadas

Veja, no vídeo a seguir, a implementação de listas encadeadas simples, duplamente encadeadas e circulares em C. Confira ainda como utilizar ponteiros para criar estruturas dinâmicas, flexíveis e eficientes para manipulação de dados.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Listas encadeadas são uma estrutura de dados dinâmica, em que cada elemento (**nó**) contém uma informação e um **ponteiro** que indica onde está o próximo elemento, conforme a imagem a seguir.

> Conteúdo interativo
>
> Acesse a versão digital para ver mais detalhes da imagem abaixo.



Lista encadeada.

Diferentemente dos vetores (ou listas estáticas), os elementos de uma lista encadeada **não precisam estar armazenados em posições contíguas da memória**. Isso oferece grande flexibilidade, em especial, em situações nas quais inserções e remoções de elementos são frequentes.

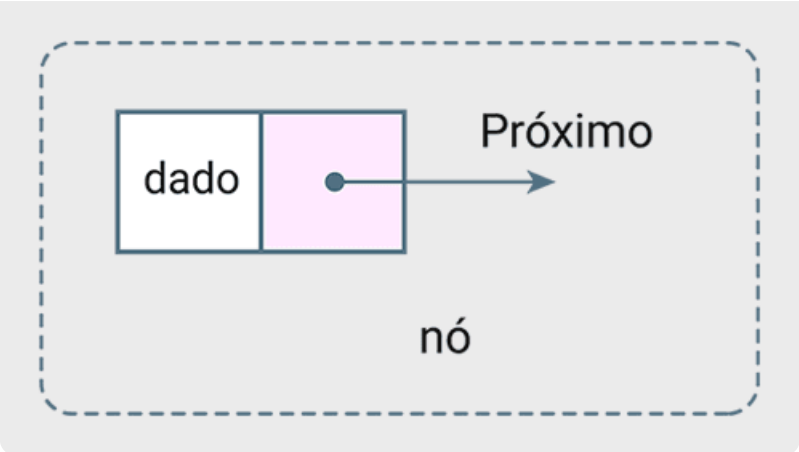
Estrutura das listas encadeadas

Uma **lista encadeada simples** é composta por nós que apontam para o próximo elemento da sequência. Cada nó contém:

- Um campo de **dados** (por exemplo, um número ou uma string).
- Um campo de **ponteiro** para o próximo nó.

Acompanhe!

> Conteúdo interativo
>
> Acesse a versão digital para ver mais detalhes da imagem abaixo.



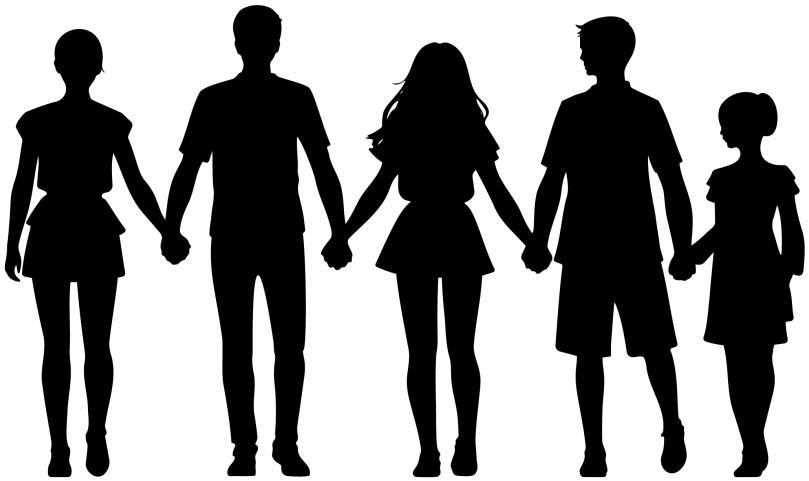
Representação do nó.

A estrutura básica em C pode ser representada assim:

```c
struct No {
    int dado;
    struct No* proximo;
};
```

Imagine que você está em uma **fila de pessoas**, e que cada uma segura a mão da próxima, como podemos ver na imagem adiante.

> Conteúdo interativo
>
> Acesse a versão digital para ver mais detalhes da imagem abaixo.



Fila de pessoas.

A posição de cada pessoa não é importante — elas só precisam estar conectadas umas às outras. Se alguém no meio da fila sair, basta que os vizinhos soltem a mão dela e deem as mãos de novo.

Essa é a ideia da **lista encadeada**: não importa onde o nó está na memória, mas, sim, que ele saiba para onde apontar.

Essa abordagem é diferente de um vetor, em que, na memória, todos os elementos estão enfileirados em ordem, como em uma estante de livros com espaços fixos. Se você quiser adicionar um novo livro no meio, terá que deslocar os outros.

Exemplo prático em C

Vamos criar manualmente uma lista com três nós com os valores 10, 20 e 30. Cada nó aponta para o próximo até que o último (terceiro) aponte para NULL, indicando o fim da lista. Veja!

```c
#include
#include

struct No {
    int dado;
    struct No* proximo;
};

int main() {
    struct No* primeiro = (struct No*) malloc(sizeof(struct No));
    struct No* segundo = (struct No*) malloc(sizeof(struct No));
    struct No* terceiro = (struct No*) malloc(sizeof(struct No));

    primeiro->dado = 10;
    primeiro->proximo = segundo;

    segundo->dado = 20;
    segundo->proximo = terceiro;

    terceiro->dado = 30;
    terceiro->proximo = NULL;

    // Percorrendo a lista
    struct No* atual = primeiro;
    while (atual != NULL) {
        printf("%d\n", atual->dado);
        atual = atual->proximo;
    }

    return 0;
}
```

Confira adiante algumas práticas importantes para garantir o uso seguro e eficiente da memória em C.

Atenção

Verifique sempre se malloc retornou NULL antes de usar o ponteiro. Libere a memória com free quando os nós não forem mais necessários para evitar vazamento de memória. Usar corretamente os ponteiros evita erros, como acessar áreas inválidas da memória (isto é, partes da memória que o programa não tem permissão para acessar).

Lista duplamente encadeada

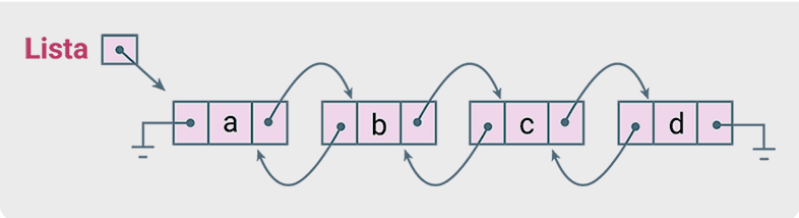
É uma estrutura de dados dinâmica, em que cada elemento, chamado de nó, contém três partes:

1. Um campo para armazenar o dado.
2. Um ponteiro para o próximo nó.
3. Outro ponteiro para o nó anterior.

Confira mais detalhes na imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Lista duplamente encadeada.

A estrutura da lista estudada permite que a navegação na lista seja feita tanto do início para o fim como do fim para o início. Ao contrário da lista encadeada simples, que só permite avançar, a lista duplamente encadeada oferece mais flexibilidade, facilitando operações como inserção e remoção de elementos em qualquer posição da lista.

Estrutura das listas duplamente encadeadas

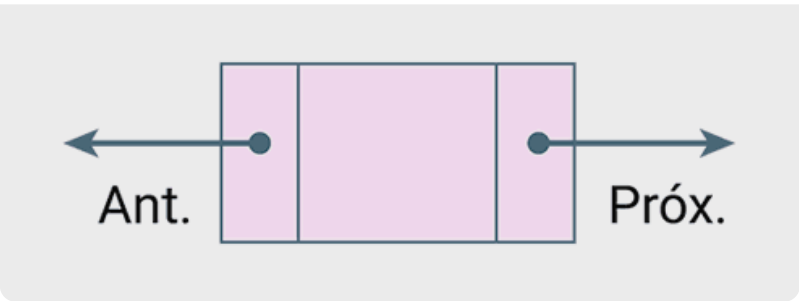
É composta por nós, e cada um deles possui:

- Um ponteiro para o **próximo** nó.
- Um ponteiro para o **nó anterior**.
- Um campo de **dados** (por exemplo, um número ou uma string).

A imagem adiante exemplifica o nó de uma lista duplamente encadeada; observe!

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Nó de uma lista duplamente encadeada.

A estrutura básica em C pode ser representada assim:

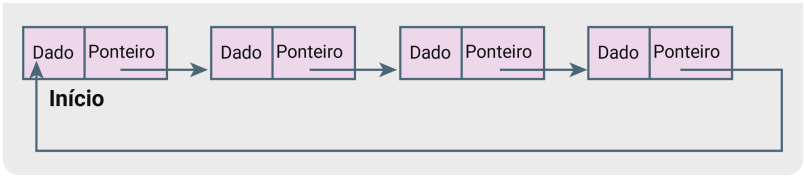
```c
struct No {
    int dado;
    struct No* proximo;
    struct No* anterior;
};
```

Lista circular simples

É uma variação da lista encadeada, em que o último nó da estrutura não aponta para NULL, mas para o primeiro nó da lista, formando um ciclo fechado. Confira na imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Lista circular simples.

Tal estrutura permite que a navegação continue indefinidamente a partir de qualquer ponto, sem um fim definido de maneira clara. Cada nó da lista circular simples contém um valor e um ponteiro para o próximo nó, assim como em uma lista encadeada comum, com a diferença de que o laço de ligação nunca se rompe.

Estrutura da lista circular simples

É composta por nós, em que o último nó aponta de volta para o primeiro, formando um ciclo.

A estrutura básica em C pode ser representada assim:

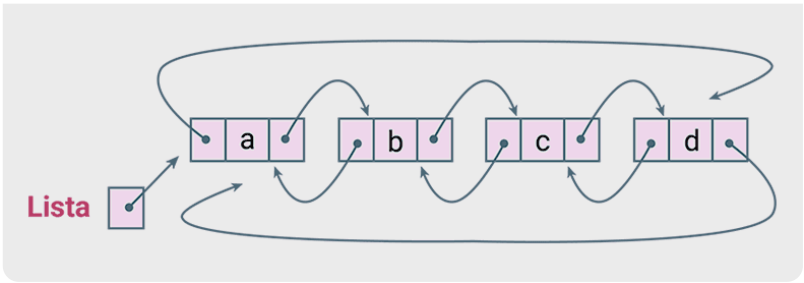
```c
struct No {
    int dado;
    struct No* proximo; // o último aponta para o primeiro
};
```

Lista circular duplamente encadeada

É uma estrutura de dados em que cada nó possui três componentes: o dado armazenado, um ponteiro para o próximo nó e um ponteiro para o nó anterior, conforme a imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Lista circular duplamente encadeada.

Diferentemente das listas comuns, ela é circular, ou seja, o último nó aponta para o primeiro, e o primeiro, para o último, criando um ciclo fechado em ambas as direções.

Estrutura da lista circular duplamente encadeada

Combina os dois conceitos, ou seja, ela é duplamente encadeada e circular. O último nó aponta para o primeiro, e o primeiro aponta para o último.

A estrutura básica em C pode ser representada assim:

```c
struct No {
    int dado;
    struct No* proximo;
    struct No* anterior;
};
```

Operações em listas: inserção, remoção e percorrimento

No vídeo a seguir, veja como inserir, remover e percorrer elementos em listas lineares e encadeadas. Acompanhe ainda como manipular estruturas dinamicamente em C e manter a integridade dos dados em memória.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

As listas, estáticas ou encadeadas, tornam-se, de fato, úteis quando conseguimos manipular seus elementos de forma eficiente. Para isso, precisamos dominar três operações fundamentais: inserção, remoção e percorrimento. Esses procedimentos estão no coração da lógica de qualquer estrutura de dados dinâmica e são muito utilizados em sistemas de cadastro, gerenciadores de tarefas, jogos e muito mais!

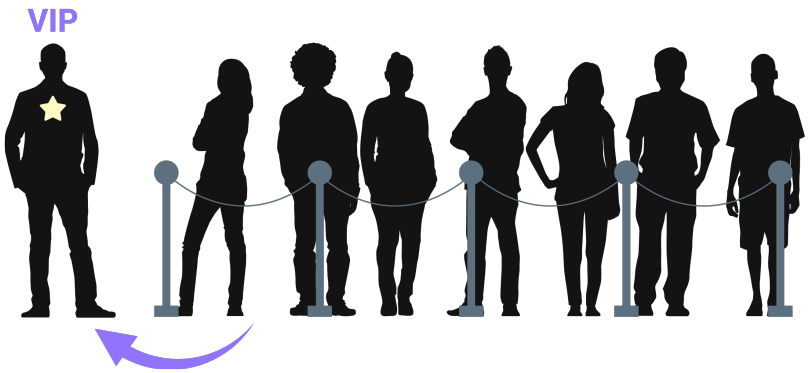
Inserção: adicionando elementos

Em listas, inserir um elemento significa colocá-lo na posição desejada na lista. Dependendo da estrutura usada (vetor ou lista encadeada), esse processo pode ser mais ou menos custoso.

Vamos entender melhor! Imagine uma fila no cinema. Quando alguém chega, é comum que vá para o final da fila — essa é uma inserção simples. Mas, se for alguém com ingresso VIP, talvez vá para a frente — o que obriga os outros a “se moverem”, como podemos observar na imagem adiante.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Fila do cinema com uma pessoa VIP.

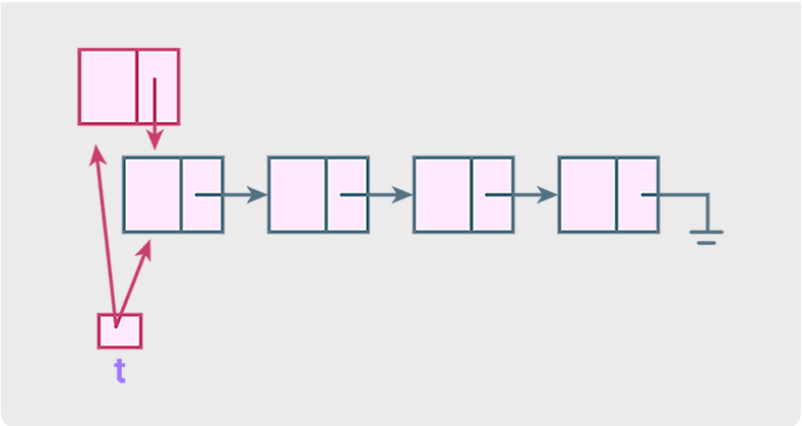
Com vetores, inserir no início ou no meio exige deslocar elementos. Já em listas encadeadas, basta ajustar os ponteiros, sem precisar mover dados de lugar.

Exemplo prático em C

Vamos criar uma lista encadeada e inserir novos nós (elementos) no início da lista. A cada inserção, o novo nó se torna o primeiro da lista, e os anteriores são "empurrados" para frente, conforme a imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Inserção no início da lista.

Para você compreender melhor como funciona a inserção no início de uma lista encadeada, observe o trecho de código em linguagem C a seguir:

```c
struct No {
    int dado;
    struct No* proximo;
};

void inserirNoInicio(struct No** inicio, int valor) {
    struct No* novo = (struct No*) malloc(sizeof(struct No));
    novo->dado = valor;
    novo->proximo = *inicio;
    *inicio = novo;
}
```

O programa implementa a **inserção no início de uma lista encadeada simples**. Isso é muito útil quando desejamos construir uma lista de forma rápida e eficiente, adicionando elementos no topo sem precisar percorrer toda a estrutura. O raciocínio foi construído nos seguintes passos:

1. Alocar memória para um novo nó.
2. Preencher esse nó com o valor recebido.
3. Fazer o novo nó apontar para aquele que era o primeiro da lista.
4. Atualizar o ponteiro da lista para que o novo nó seja o novo início.

Remoção: excluindo elementos

Além de eliminar o nó da lista, removê-lo também significa reorganizar a estrutura de modo que a lista continue funcional. Em listas encadeadas, isso é feito ajustando os ponteiros dos nós vizinhos. Já em vetores, é preciso deslocar todos os elementos seguintes uma posição para trás.

Exemplo

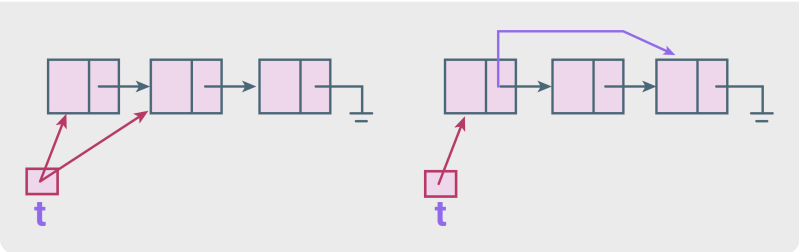
Pensando ainda em filas, imagine que alguém decide sair de uma. Se ela estiver em um grupo com pessoas de mãos dadas, bastaria que os dois ao lado dessem as mãos de novo. Em vetores, seria como mover todas as pessoas um passo à frente para tapar o espaço.

Exemplo prático em C

Definimos uma função chamada removerDoInicio, cujo objetivo é **remover o primeiro elemento de uma lista encadeada simples**. Ela lida com a manipulação dinâmica da memória e o ajuste dos ponteiros para manter a integridade da lista, como pode ser visto a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Remoção.

Para compreender esse conceito na prática, veja um exemplo de função em C que remove um elemento do início de uma lista encadeada:

```c
void removerDoInicio(struct No** inicio) {
    if (*inicio != NULL) {
        struct No* temp = *inicio;
        *inicio = (*inicio)->proximo;
        free(temp);
    }
}
```

Trata-se de uma operação muito comum em estruturas como filas e listas ligadas, em que a inserção é feita no início, ao passo que a remoção ocorre no início ou no final. Lembre-se: saber fazer isso de maneira adequada evita vazamentos de memória e erros de ponteiros. Então, o raciocínio foi construído nos seguintes passos:

1. Verificar se a lista tem, pelo menos, um elemento.
2. Armazenar o ponteiro para o primeiro nó.
3. Fazer a lista apontar para o segundo nó.
4. Liberar a memória ocupada pelo primeiro nó.

Navegação: visitando cada elemento

Quando falamos em navegação em uma fila, precisamos entender que se trata do processo de passar por cada um de seus elementos. Em geral, o objetivo é exibir os dados, buscar um valor ou aplicar alguma operação.

Pensando ainda no exemplo da fila do cinema, agora é hora de fazer uma inspeção para verificar quem está com o ingresso. É preciso ir de um em um, até o fim.

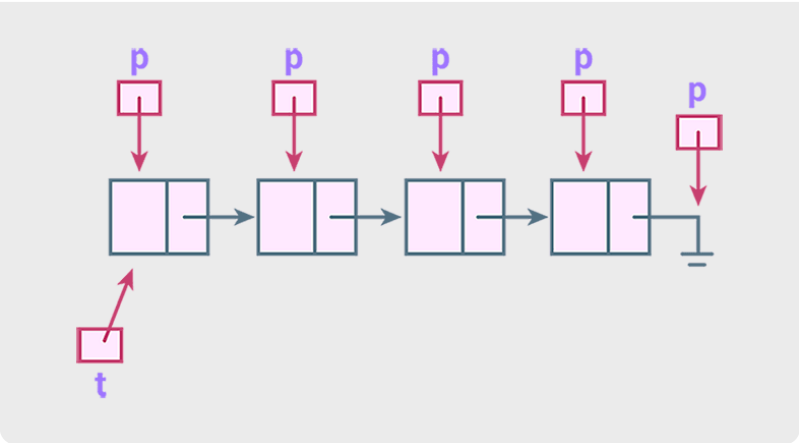
Exemplo prático em C

Vamos definir uma função chamada listar, cujo objetivo é percorrer (ou iterar) uma **lista encadeada simples** e **exibir os valores armazenados em cada nó**. Além disso, temos também o propósito de navegar do início ao fim

da lista encadeada, **imprimindo na tela o conteúdo de cada nó**. Essa operação ajuda a **visualizar os dados armazenados** na estrutura, como mostra a imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Listagem.

A seguir, veja um exemplo de função usada para percorrer uma lista encadeada exibindo os seus elementos:

```c
void listar(struct No* inicio) {
    struct No* atual = inicio;
    while (atual != NULL) {
        printf("%d\n", atual->dado);
        atual = atual->proximo;
    }
}
```

A função listar é uma forma de **visualizar os elementos** de uma lista encadeada. Ela é essencial para testes, depuração e para que o usuário veja os dados armazenados. Além disso, ela ainda demonstra o conceito de **navegação (iteração)** em uma estrutura ligada, base para operações mais avançadas, como busca, ordenação e manipulação de dados.

Busca linear

O vídeo a seguir ensina como implementar a busca linear em vetores e listas encadeadas, explicando sua lógica, aplicabilidade em listas não ordenadas e desempenho proporcional ao número de elementos da estrutura.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Também conhecida como busca sequencial, a busca linear é uma das formas mais simples e diretas de procurar um elemento em uma lista. Ela consiste em examinar, um por um, todos os elementos da estrutura

até encontrar o valor desejado ou atingir o fim da lista. Apesar de ser uma técnica básica, a busca linear é muito importante por sua aplicabilidade em listas não ordenadas e pela clareza de sua lógica.

A ideia da busca linear é simples: **verificar cada elemento da lista até achar o que se procura**. Isso significa que a busca percorre a lista do início ao fim (ou até encontrar o elemento), comparando cada item com o valor alvo.

Vamos entender melhor! Imagine que você está procurando uma chave em uma bolsa cheia de objetos. Praticamente uma missão impossível, certo? Como os itens não estão organizados, você precisa olhar um por um até achar a chave. Esse é exatamente o funcionamento da busca linear!

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

Bolsa com objetos.

Outro exemplo: em uma lista de chamada não ordenada, o professor precisa ler todos os nomes até encontrar o do aluno que está sendo chamado.

Exemplo prático em C (vetor)

Neste caso, vamos:

- Percorrer o vetor lista até o índice tamanho.
- Comparar cada elemento com o valor buscado.
- Encontrando, retornar à posição.
- Não encontrando, retornar -1, indicando que o valor não está presente.

Com base nesses passos, temos a implementação da busca linear em um vetor, utilizando a linguagem C:

```c
int buscaLinear(int lista[], int tamanho, int valor) {
    for (int i = 0; i < tamanho; i++) {
        if (lista[i] == valor) {
            return i; // Retorna o índice onde encontrou o valor
        }
    }
    return -1; // Retorna -1 se não encontrou
}
```

Exemplo prático em C (lista encadeada)

Vamos percorrer uma lista encadeada, nó por nó, verificando se o campo dado é igual ao valor procurado. A variável pos ajuda a identificar em qual posição da lista o valor foi encontrado. Confira!

```c
int buscaLinearLista(struct No* inicio, int valor) {
    int pos = 0;
    struct No* atual = inicio;
    while (atual != NULL) {
        if (atual->dado == valor) {
            return pos;
        }
        atual = atual->proximo;
        pos++;
    }
    return -1;
}
```

Desempenho

Como podemos ver na tabela a seguir, no **pior caso**, todos os elementos precisam ser verificados.

pior caso

O pior caso em uma busca linear ocorre quando o elemento procurado está na última posição da estrutura ou não está presente, pois, nesse caso, o algoritmo precisará verificar todos os elementos.

Situação	Número médio de comparações
Valor está no início	1
Valor está no meio	$n / 2$
Valor está no final	n
Valor não está na lista	n

Tabela: número médio de comparações por situação.
Curadoria de TI.

Lembre-se: na tabela, n é o número de elementos.

Busca manual X busca linear

Imagine que você recebeu uma caixa com 50 chaves misturadas e precisa encontrar a do carro. Como a caixa está desorganizada, sua melhor estratégia é pegar chave por chave, olhando uma a uma, até encontrar a correta.

Esse processo é **sequencial**, **manual** e pode ser cansativo — afinal, quanto mais chaves, mais demorado. Esse método equivale à **busca linear** usada em algoritmos.

Exemplo

Um caso clássico é a busca por um nome em uma lista de presença impressa, fora de ordem alfabética. Você começa lendo de cima para baixo até encontrar o nome. Cada linha lida é uma comparação. E se o nome não estiver lá? Você vai ter que chegar ao fim da lista para descobrir!

Como o computador realiza a busca?

Em C, a lógica da busca é implementada com estruturas como vetores e listas. A busca sequencial automatiza o mesmo comportamento humano de "olhar item por item". Veja um exemplo com vetor:

```c
int buscaLinear(int lista[], int tamanho, int alvo) {
    for (int i = 0; i < tamanho; i++) {
        if (lista[i] == alvo) {
            return i;
        }
    }
    return -1; // Não encontrado
}
```

O código imita o processo manual de olhar elemento por elemento. Mas, diferentemente de nós, o computador realiza essas comparações muito mais rápido — afinal, são milhares por segundo.

Por fim, a busca linear é uma técnica essencial, em especial quando lidamos com listas **não ordenadas**. Ela é fácil de implementar, compreensível até para iniciantes e funciona para vetores para listas encadeadas. Embora não seja a mais eficiente para grandes volumes de dados, ela é uma ótima escolha para estruturas pequenas ou quando não há garantias sobre a ordem dos elementos.

Hora de codar

Confira o vídeo a seguir, que apresenta o desenvolvimento de um sistema de inventário em C, comparando vetor e lista encadeada, com inserção, remoção, listagem e busca linear, explorando os limites, a flexibilidade e o desempenho de cada estrutura.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Você está desenvolvendo um jogo de aventura, e cada jogador possui uma mochila que pode ser gerenciada de duas formas:

Lista estática (vetor)

Para situações com espaço fixo.

Lista encadeada

Para flexibilidade total.

O jogador poderá:

- Inserir novos itens
- Remover itens existentes
- Listar todos os itens
- Buscar um item pelo nome

O objetivo é implementar um minissistema de gerenciamento de itens que utiliza:

- Vetor (lista estática)
- Lista encadeada
- Inserção
- Remoção
- Listagem de itens
- Busca linear
- Comparação entre busca manual (humana) e automatizada.

Desafio: novato

Assista, no vídeo a seguir, ao desenvolvimento da lógica do sistema de inventário inicial, fundamental para a sobrevivência do jogador logo após pousar na ilha.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O que você vai fazer?

Neste nível, o jogador precisa coletar rapidamente recursos essenciais (armas, munições, kits médicos e ferramentas) e organizar esses itens dentro da mochila virtual, garantindo eficiência nas primeiras decisões estratégicas do jogo. Para isso, você implementará, com o uso de estruturas de dados compostas (structs) e listas sequenciais, a representação dos itens do inventário.

Sua missão é construir um sistema de inventário que simule a **mochila de loot inicial** do jogador. Para isso, você criará uma struct chamada Item, que armazenará informações essenciais de cada objeto coletado. O sistema permitirá que o jogador cadastre, remova, liste e busque por itens dentro da mochila.

Requisitos funcionais

Confira as principais funcionalidades do sistema a serem implementadas:

1. **Criação da struct:** definir uma struct chamada Item com os campos char nome[30], char tipo[20] e int quantidade.
2. **Cadastro de itens:** o sistema deve permitir que o jogador cadastre até 10 itens informando nome, tipo (ex: arma, munição e cura) e quantidade.
3. **Remoção de itens:** permitir que o jogador exclua um item da mochila, informando seu nome.
4. **Listagem dos itens registrados com seus dados:** o que deve ocorrer após cada operação.
5. **Busca sequencial:** implementar uma função de busca sequencial que localize um item na mochila com base no nome e exiba seus dados.

Requisitos não funcionais

Considere também os seguintes critérios durante o desenvolvimento:

1. **Usabilidade:** a interface do sistema deve ser clara e orientativa, com mensagens que auxiliem o jogador nas decisões.
2. **Desempenho:** o sistema deve responder a cada comando com menos de 2 segundos.
3. **Documentação:** o código deve conter comentários explicativos sobre a criação da struct, o uso das funções e o fluxo de execução.
4. **Manutenibilidade:** uso de nomes de variáveis e funções que representem com clareza sua função, facilitando futuras alterações no código.

Instruções detalhadas

A seguir, veja os elementos básicos que devem compor a estrutura do programa:

1. **Bibliotecas necessárias:** inclua stdio.h, string.h, e stdlib.h.

2. **Definição da struct:** declare a struct Item com os campos solicitados.

3. **Vetor de structs:** crie um vetor com capacidade para armazenar até 10 itens.

4. **Funções obrigatórias:** inserir Item(); removerItem(); listarItens(); buscarItem()

5. **Leitura de dados:** use scanf() para ler valores inteiros e fgets() ou scanf("%s", ...) para capturar strings com cuidado.

6. **Uso de laços:** use for ou while para manipular e percorrer o vetor de itens.

Comentários adicionais

Este desafio introduz o conceito de structs como agregadores de dados e vetores como estruturas sequenciais, baseando-se em cenários reais de jogos de sobrevivência. Ao final da atividade, você terá construído um sistema básico de inventário aplicando conceitos fundamentais de organização de dados, modularização com funções e busca sequencial, abrindo caminho para estruturas mais avançadas nos próximos níveis do jogo.

Entregando seu projeto

Finalize sua entrega seguindo os próximos passos, para que tudo esteja corretamente versionado e disponível no GitHub:

1. **Desenvolva seu projeto no GitHub:** use sempre o mesmo repositório do GitHub.

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:** mande 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, garantindo 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

Acesse o 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. Atenção: é esse link que você deve enviar no SAVA.

Explore a estrutura do ambiente GitHub

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.

Para finalizar este nível, comente todos os arquivos de código-fonte, pois isso demonstra o quanto você saber 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.

Confira o vídeo a seguir, com dicas e detalhes sobre a entrega do seu projeto. Imperdível!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Manipulação por encadeamento X estruturas sequenciais

O vídeo a seguir compara vetores e listas encadeadas em C, analisando as diferenças de acesso, inserção, remoção e uso de memória. Entenda qual estrutura é mais adequada para cada tipo de aplicação. Vamos lá!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Quando estudamos estruturas de dados, precisamos entender como eles são manipulados internamente — em especial, ao compararmos duas abordagens clássicas: a **manipulação sequencial (com vetores)** e a **manipulação por encadeamento (com listas ligadas)**. Embora armazenem coleções de elementos, como elas gerenciam, acessam e reorganizam esses dados é bastante diferente — e essas diferenças impactam a eficiência dos algoritmos por completo!

Estrutura sequencial – vetores

Primeiro, vamos entender que vetores são estruturas de dados sequenciais e estáticas. Isso significa que os elementos são armazenados em posições contíguas na memória, e o número total de posições deve ser definido com antecipação.

Exemplo prático em C

Nesse tipo de estrutura, acessar um elemento específico é muito rápido, pois o índice do vetor permite o acesso direto. Porém, inserir ou remover elementos **no meio** da estrutura exige o deslocamento dos elementos seguintes. Isso pode ser custoso, sobretudo em listas longas. Observe o código adiante.

```c
int v[5] = {10, 20, 30, 40, 50};
printf("%d", v[2]); // imprime 30
```

Estrutura encadeada – listas ligadas

As listas encadeadas usam um modelo dinâmico, em que cada elemento (ou nó) contém:

- O dado em si.
- Um ponteiro para o próximo nó.

Exemplo prático em C

Ao contrário dos vetores, os nós de uma lista encadeada não estão necessariamente em posições adjacentes na memória. A ligação entre os nós se dá por ponteiros, o que permite inserções e remoções rápidas em qualquer ponto da lista. Acompanhe a seguir!

```c
struct No {
    int valor;
    struct No* proximo;
};
```

Pense em um armário com gavetas numeradas (vetor). Para acessar a gaveta 3, você vai direto nela. Mas se quiser inserir algo entre a gaveta 2 e a 3, você precisa empurrar todas as gavetas seguintes para abrir espaço — e isso consome tempo, não é mesmo?

Exemplo

Imagine uma corrente de elos (lista encadeada). Se você quiser inserir um novo elo entre dois existentes, basta abrir a corrente e conectar o novo elo. Isso é muito mais flexível, especialmente quando não sabemos quantos elos serão necessários.

Aplicações das manipulações sequenciais e por encadeamento

Na prática da programação, a decisão entre utilizar uma estrutura sequencial ou uma estrutura encadeada está relacionada ao tipo de problema que se pretende resolver. Cada abordagem oferece vantagens específicas de acordo com fatores (como desempenho, necessidade de flexibilidade e frequência das operações de inserção ou remoção).

A seguir, veja exemplos de situações em que uma ou outra opção pode ser mais indicada.

Manipulação sequencial

Ideal para acesso rápido e listas de tamanho conhecido (exemplo: vetor de temperaturas de uma semana).

Manipulação por encadeamento

Recomendada para casos em que o tamanho da lista varia ou há muitas inserções/remoções no meio (exemplo: sistema de filas dinâmicas, listas de tarefas e buffers).

Contrastando a manipulação

Para entender melhor as diferenças entre as estruturas sequenciais e encadeadas, é útil observar como cada uma se comporta nas principais operações de manipulação. A tabela a seguir compara essas abordagens, evidenciando pontos fortes e limitações em critérios como desempenho, flexibilidade e uso de memória.

Operação	Estrutura sequencial (vetor)	Estrutura encadeada (lista)
Acesso por posição	Rápido	Lento
Inserção no meio	Lenta – desloca dados	Rápida – ajusta ponteiros
Remoção no meio	Lenta – desloca dados	Rápida – ajusta ponteiros
Crescimento da estrutura	Limitado ao tamanho fixo	Flexível – cresce conforme uso

Operação	Estrutura sequencial (vetor)	Estrutura encadeada (lista)
Uso de memória	Contíguo e previsível	Fragmentado e dinâmico

Tabela: Comparação das principais operações em vetores e listas encadeadas.
Curadoria de TI.

Compreender a diferença entre manipulação por encadeamento e manipulação sequencial é fundamental para projetar algoritmos mais eficientes.

Além disso, cada abordagem tem seus pontos fortes:

- Vetores são excelentes para acesso direto e simplicidade.
- Listas encadeadas são melhores em flexibilidade e manipulação dinâmica.

Busca binária

O vídeo a seguir explora a busca binária em vetores ordenados, explicando sua lógica de divisão e conquista, bem como a implementação em C, o desempenho $O(logn)$ e a comparação com a busca sequencial em termos de eficiência.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Busca binária é um algoritmo bastante eficiente para localizar elementos em estruturas de dados ordenadas. Diferentemente da busca sequencial, que verifica cada item, um por um, a busca binária divide o problema ao meio a cada passo, eliminando metade dos elementos restantes em cada iteração. Essa estratégia reduz (e muito!) o número de comparações, sobretudo em listas grandes, e é um exemplo clássico do paradigma de divisão e conquista.

Conceito de busca binária

A ideia é simples: dado um vetor ordenado em ordem crescente, começamos comparando o elemento do meio com o valor que procuramos. Se o valor for igual, o elemento foi encontrado. Se for menor, descartamos a metade superior; e se for maior, descartamos a metade inferior. Esse processo se repete até encontrarmos o valor ou concluirmos que ele não está presente, conforme a imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

Busca binária do número 21.

Imagine que você está procurando uma palavra no dicionário impresso. Você não começa na primeira página — você vai direto para o meio, compara, e, então, decide se deve olhar mais para frente ou para trás. Aliás, cada vez que você abre o dicionário, reduz bastante o número de páginas possíveis. Isso é busca binária em ação!

Exemplo prático em C

Veja a seguir a implementação da busca binária na forma iterativa, utilizada para encontrar a posição de um determinado valor em um vetor ordenado.

A função recebe três parâmetros:

- O vetor onde será feita a busca.
- O tamanho do vetor.
- O valor desejado.

Se o valor for encontrado, a função retorna sua posição no vetor; caso contrário, retorna -1, indicando que o valor não está presente.

```c
int buscaBinaria(int vetor[], int tamanho, int valor) {
    int inicio = 0, fim = tamanho - 1;

    while (inicio <= fim) {
        int meio = (inicio + fim) / 2;

        if (vetor[meio] == valor)
            return meio;
        else if (vetor[meio] < valor)
            inicio = meio + 1;
        else
            fim = meio - 1;
    }

    return -1; // valor não encontrado
}
```

No código, vemos que:

- As variáveis inicio e fim delimitam os extremos da área que ainda estamos considerando.
- A cada passo, atualizamos inicio ou fim dependendo da comparação com o elemento central (meio).

Desempenho da busca binária

Diferentemente da busca sequencial, a binária é muito mais eficiente, pois, a cada iteração, o número de elementos restantes é dividido por dois. Isso significa que em uma lista com 1.000 elementos, no máximo 10 comparações serão necessárias, pois ($2^{10} = 1024$). Já a busca linear poderia exigir até 1.000 comparações no pior caso.

Requisitos importantes

Como nem tudo são flores, atente-se: a busca binária só funciona se os **dados estiverem ordenados**. Portanto, se o vetor estiver desorganizado, o algoritmo produzirá resultados incorretos.

Apesar disso, a busca binária é um dos algoritmos mais eficientes e elegantes da ciência da computação. Ao explorar o princípio da divisão e conquista, ela mostra como a estrutura e a estratégia de busca fazem toda a diferença no desempenho de um programa.

Implementação recursiva e iterativa

Veja, no vídeo a seguir, a diferença entre as abordagens recursiva e iterativa em C, analisando vantagens, desvantagens e a aplicação prática na busca binária para melhorar clareza, desempenho e uso de memória.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Na programação, muitos algoritmos podem ser escritos de forma **iterativa** (usando laços for ou while) ou de forma **recursiva** (funções que chamam a si mesmas). Compreender essas abordagens ajuda a resolver problemas de forma mais adequada e eficiente.

Vamos explorar a diferença entre os dois estilos de implementação e como cada um impacta na estrutura e execução do código — com destaque para a busca binária, um clássico que pode ser implementado dos dois modos.

Implementação iterativa

É baseada em laços de repetição. O algoritmo mantém controle das variáveis (como início e fim) e as atualiza a cada passo, até que a condição de parada seja satisfeita.

Vantagens

- Mais eficiente em termos de uso de memória (evita empilhamento de chamadas).
- Costuma ser mais rápida na execução em baixo nível.
- Mais próxima da forma como o computador executa instruções.

Exemplo prático em C

Veja a implementação da versão iterativa da busca binária, um algoritmo eficiente para encontrar um elemento em um vetor já ordenado em ordem crescente.

A função recebe três parâmetros:

- O vetor onde será feita a busca.
- O tamanho do vetor.
- O valor procurado.

Utilizando duas variáveis (inicio e fim) para definir os limites da busca e uma variável meio para dividir o vetor ao meio a cada iteração, o algoritmo compara progressivamente o valor do meio com o desejado, descartando metade do vetor a cada passo. Caso o valor seja encontrado, a função retorna sua posição; caso contrário, retorna -1, indicando que o elemento não está presente no vetor.

```c
int buscaBinaria(int vetor[], int tamanho, int valor) {
    int inicio = 0, fim = tamanho - 1;

    while (inicio <= fim) {
        int meio = (inicio + fim) / 2;

        if (vetor[meio] == valor)
            return meio;
        else if (vetor[meio] < valor)
            inicio = meio + 1;
        else
            fim = meio - 1;
    }

    return -1; // não encontrado
}
```

Implementação recursiva

Resolve o problema dividindo-o em subproblemas menores, chamando a própria função com novos parâmetros. A cada chamada, uma nova instância da função é empilhada até que a condição de parada seja alcançada.

Confira mais detalhes!

Vantagens

- Trata-se de um código mais limpo e elegante.
- É ideal para problemas naturalmente recursivos (como árvores, grafos ou fatorial).

Desvantagens

- Pode consumir mais memória, pois cada chamada ocupa espaço na pilha.
- Pode causar "estouro de pilha" se não houver condição de parada ou se a profundidade de chamadas for muito grande.

Exemplo prático em C

Vamos agora conferir a implementação da versão recursiva da busca binária, um algoritmo utilizado para localizar um valor específico em um vetor ordenado.

A função recebe como parâmetros:

- O vetor.
- Os índices que delimitam o intervalo de busca (inicio e fim).
- O valor a ser procurado.

O algoritmo compara o valor do meio do intervalo com o desejado e, com base no resultado, faz uma chamada recursiva para continuar a busca na metade inferior ou superior do vetor. Esse processo se repete até que o valor seja encontrado ou que o intervalo se torne inválido (quando inicio ultrapassa fim), o que indica que o elemento não está presente. A recursão torna o código conciso e elegante, facilitando a compreensão da lógica de divisão do vetor. Veja!

```c
int buscaBinariaRecursiva(int vetor[], int inicio, int fim, int valor) {
    if (inicio > fim)
        return -1;

    int meio = (inicio + fim) / 2;

    if (vetor[meio] == valor)
        return meio;
    else if (vetor[meio] < valor)
        return buscaBinariaRecursiva(vetor, meio + 1, fim, valor);
    else
        return buscaBinariaRecursiva(vetor, inicio, meio - 1, valor);
}
```

Imagine você procurando uma página em um livro usando um marcador. Você abre no meio, verifica, avança ou recua com as mãos. Você está presente em cada ação, repetindo fisicamente o processo. Esse é um exemplo de abordagem iterativa.

Agora, imagine que você peça a um assistente para abrir o livro no meio. Se ele não encontrar, você pede para outro repetir a busca com metade das páginas, e assim por diante. Quando alguém acha a página, todos os assistentes voltam com a resposta. Cada assistente é como uma chamada recursiva. Esse é um exemplo de abordagem **recursiva**.

Entender a diferença entre implementação iterativa e recursiva ajuda o programador a escrever códigos mais eficientes e adequados ao problema. A escolha entre um e outro depende de critérios como desempenho, legibilidade e uso de memória. Entretanto, em problemas como a busca binária, ambas as abordagens são válidas.

Busca sequencial X Busca binária

O vídeo a seguir compara a busca sequencial e a busca binária em C, explicando suas estratégias, requisitos e aplicabilidade para ajudar na escolha do algoritmo mais adequado conforme o tamanho e a ordenação dos dados. Não perca!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Buscar informações é uma das tarefas mais comuns na computação. Dois dos algoritmos mais utilizados para isso são a busca sequencial (ou linear) e a busca binária. Ambos têm o mesmo objetivo — localizar um valor em uma estrutura de dados —, mas utilizam estratégias diferentes, com custos e requisitos distintos. Compreender suas diferenças é fundamental para escolher a abordagem correta em cada situação.

Busca sequencial

Percorre os elementos de uma lista **um a um**, do início ao fim, até encontrar o valor desejado ou atingir o final da estrutura.

Exemplo prático em C

A seguir, temos a implementação da busca sequencial em linguagem C. Note que se o valor não for encontrado, o resultado retornado será -1, indicando ausência no vetor. Veja como isso é feito na prática:

```c
int buscaSequencial(int vetor[], int tamanho, int valor) {
    for (int i = 0; i < tamanho; i++) {
        if (vetor[i] == valor) {
            return i;
        }
    }
    return -1;
}
```

Busca binária

Características principais

A seguir, listamos as mais importantes; confira!

- Não exige que os dados estejam ordenados.
- É simples de implementar e entender.
- Funciona para qualquer tipo de estrutura linear (vetores, e listas).

Mas atenção: a busca binária só pode ser utilizada em **estruturas ordenadas**. Ela funciona dividindo o espaço de busca ao meio a cada iteração (ou chamada recursiva), comparando o valor do meio com o procurado e descartando metade dos dados a cada passo.

Exemplo prático em C

Veja a seguir a implementação da função de busca binária em linguagem C:

```c
int buscaBinaria(int vetor[], int tamanho, int valor) {
    int inicio = 0, fim = tamanho - 1;
    while (inicio <= fim) {
        int meio = (inicio + fim) / 2;
        if (vetor[meio] == valor)
            return meio;
        else if (vetor[meio] < valor)
            inicio = meio + 1;
        else
            fim = meio - 1;
    }
    return -1;
}
```

A busca binária possui algumas características importantes:

- Exige que os dados já estejam ordenados, seja em ordem crescente ou decrescente (com as devidas adaptações).
- É bastante eficiente, especialmente quando aplicada a listas grandes.

Imagine procurar o nome de alguém em uma lista de presença embaralhada. Você precisa ler linha por linha até encontrar. Esse é um exemplo de **busca sequencial.**

Agora imagine que essa mesma lista está em ordem alfabética. Você pode ir direto para o meio, ver se o nome está acima ou abaixo na ordem, e repetir esse processo — economizando tempo a cada passo. Esse é um exemplo de **busca binária**.

Comparativo entre busca sequencial e binária

Acompanhe o quadro adiante, que destaca as principais diferenças entre a **busca sequencial** e a **busca binária**, facilitando a escolha da melhor abordagem conforme o contexto.

Busca sequencial	Busca binária
Destaca-se pela **simplicidade na implementação** e **versatilidade**, já que funciona em qualquer lista, mesmo que não esteja ordenada, e pode ser aplicada até em listas encadeadas. No entanto, seu desempenho se torna ineficiente em listas grandes, já que exige uma comparação por elemento.	É **mais eficiente em termos de desempenho**, sobretudo em grandes volumes de dados. Em contrapartida, ela exige que os dados estejam **previamente ordenados** e não é tão adequada para listas encadeadas, em que o acesso direto aos elementos não é possível.

A escolha entre os dois métodos deve considerar a estrutura utilizada, o tamanho do conjunto de dados e a necessidade de desempenho. Agora, observe a tabela a seguir.

Critério	Busca sequencial	Busca binária
Dados precisam estar ordenados?	Não	Sim

Critério	Busca sequencial	Busca binária
Facilidade de implementação	Muito fácil	Um pouco mais complexa
Desempenho em listas pequenas	Suficiente	Muito bom
Desempenho em listas grandes	Lento	Rápido
Aplicável a listas encadeadas	Sim	Não ideal

Tabela: Comparação entre as buscas sequencial e binária.
Curadoria de TI

Tanto a **busca sequencial** quanto a **busca binária** têm seus méritos e devem ser escolhidas conforme o contexto. Se os dados não estão ordenados ou a lista é pequena, a busca sequencial é simples e eficiente. A busca binária, por sua vez, é ideal para grandes volumes de dados já ordenados, oferecendo desempenho muito superior.

Hora de codar

Assista ao vídeo a seguir, que mostra o desenvolvimento de um sistema em C para gerenciar um ranking de jogadores, com inserção, remoção, buscas diversas e comparação entre vetores e listas, avaliando o desempenho de cada estrutura. Imperdível!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Você está criando o protótipo de um sistema para armazenar e consultar os jogadores de um campeonato online. Os jogadores são organizados em um ranking baseado em pontuação (ordem decrescente).

O objetivo é desenvolver um sistema que gerencie pontuações de jogadores em um ranking, permitindo:

- Inserção e remoção de jogadores.
- Visualização do ranking.
- Busca por jogador (nome).
- Comparação entre vetores e listas encadeadas.
- Teste prático entre busca sequencial e binária (iterativa e recursiva).

O sistema deve aceitar:

- Cadastro e remoção de jogadores.
- Visualização do ranking.

- Busca por nome (com diferentes algoritmos).
- Análise comparativa do desempenho das estruturas.

Desafio: nível aventureiro

O vídeo a seguir apresenta o desenvolvimento de duas versões do sistema de mochila: uma usando vetor e outra com lista encadeada, para aprimorar o sistema de inventário. Assista!

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

O que você vai fazer?

Sua responsabilidade neste nível é comparar duas formas diferentes de organizar e acessar os dados da mochila: o uso de vetores (listas sequenciais) e listas encadeadas (estruturas dinâmicas). O objetivo é demonstrar como a escolha da estrutura de dados pode influenciar na performance do jogo, em especial, em situações de pressão, como a zona de perigo.

Sua missão é desenvolver duas versões do sistema de mochila: uma usando vetor e outra com lista encadeada. Você deverá implementar as mesmas operações básicas (inserção, remoção, listagem e busca) nas duas versões, e, ao final, comparar o desempenho das estruturas. Após ordenar os dados no vetor, você deverá aplicar uma **busca binária** para localizar com rapidez um item crítico.

Requisitos funcionais

Confira as principais funcionalidades do sistema a serem implementadas:

1. **Criação de structs**:

- Item: com os campos char nome[30], char tipo[20], int quantidade.
- No: com os campos Item dados e struct No* proximo para a lista encadeada.

2. **Implementações paralelas**:

- Mochila com vetor (lista sequencial).
- Mochila com lista encadeada.

3. **Operações obrigatórias nas duas estruturas**:

- Inserir novo item.
- Remover item por nome.

- Listar todos os itens da mochila.
- Buscar item por nome (busca sequencial).

4. **Ordenação no vetor**:

- Ordenar os itens por nome (Bubble Sort ou Selection Sort).

5. **Busca binária (apenas no vetor)**:

- Após a ordenação, permitir busca binária por nome.

6. **Contador de operações**:

- Exibir o número de comparações realizadas em cada tipo de busca (sequencial e binária).

Requisitos não funcionais

Considere também os seguintes critérios relevantes durante o desenvolvimento:

1. **Usabilidade:** o menu deve permitir ao jogador escolher entre vetor ou lista e mostrar os resultados com clareza.

2. **Eficiência:** o sistema deve apresentar tempo de resposta inferior a 2 segundos em cada operação.

3. **Documentação:** comentar o código explicando cada função e destacando as diferenças entre as estruturas.

4. **Clareza na nomenclatura:** usar nomes de variáveis e funções representativos do comportamento esperado.

Instruções detalhadas

A seguir, veja os elementos básicos que devem compor a estrutura do programa:

1. **Bibliotecas necessárias:** stdio.h, stdlib.h, string.h, time.h (opcional para medir tempo).

2. Modularização:

- Crie arquivos ou funções separadas para o vetor e a lista encadeada.
- Crie funções: inserirItemVetor, removerItemVetor, ordenarVetor, buscarSequencialVetor, buscarBinariaVetor, e equivalentes para a lista encadeada.

3. Contadores:

- Inclua variáveis globais ou ponteiros para contar o número de comparações feitas em cada tipo de busca.

Comentários adicionais

O foco deste desafio é o **desenvolvimento da habilidade de comparar estruturas de dados na prática**. Você compreenderá que vetores e listas encadeadas têm comportamentos diferentes em operações básicas e que a ordenação é fundamental para habilitar algoritmos de busca mais rápidos, como a **busca binária**. O desafio ainda introduz o conceito de **análise empírica**, aproximando a experiência da realidade enfrentada por desenvolvedores ao lidar com decisões de desempenho em jogos e aplicações críticas.

Entregando seu projeto

Finalize sua entrega seguindo os passos a seguir para que tudo esteja corretamente versionado e disponível no GitHub:

1. **Desenvolva seu projeto no GitHub:** continue usando 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 por meio 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, a fim de 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

Acesse o 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. É esse link que você deve enviar no SAVA.

Explore a estrutura do ambiente GitHub
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 este nível, 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.

Confira o vídeo a seguir, com dicas e detalhes para a entrega do seu projeto.

Conteúdo interativo
Acesse a versão digital para assistir ao vídeo.

Métodos de ordenação

O vídeo a seguir explora os algoritmos de ordenação em C, desde métodos simples até estratégias mais eficientes, destacando sua aplicação, seu desempenho e impacto na organização de dados.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Quando os dados estão organizados de forma ordenada — seja em ordem crescente ou decrescente —, torna-se mais fácil e eficiente realizar buscas, comparações e análises. Daí por que a ordenação de dados é uma tarefa fundamental na computação.

Os métodos de ordenação são algoritmos que organizam os elementos de uma estrutura (como vetores) com base em algum critério, como valor numérico, ordem alfabética ou data. Existem diversos algoritmos de ordenação, e cada um com suas características, vantagens e desvantagens. Portanto, a escolha do método adequado depende do tipo de dados, da quantidade de elementos e da necessidade de desempenho.

Por que ordenar?

Para melhorar a eficiência de outras operações, como a busca binária, que exige que os dados estejam previamente ordenados. Além disso, dados organizados tornam as informações mais claras e compreensíveis para os usuários e facilitam o armazenamento e o processamento em sistemas maiores.

Tipos de algoritmos de ordenação

Existem diversos algoritmos que realizam a tarefa de ordenar dados. Os tipos mais básicos são:

- **Bubble sort**
- **Insertion sort**
- **Selection sort**

Trata-se também de métodos fáceis de implementar e entender, mas, em geral, eles também são ineficientes para grandes volumes de dados. Eles costumam ser utilizados para fins educacionais e em situações em que o desempenho não é crítico.

Imagine que você está organizando livros em uma estante por ordem de título. Com métodos simples, você pode ir comparando dois a dois e trocando de lugar (como no bubble sort).

Exemplo prático em C

Independentemente do método, a base de um algoritmo de ordenação envolve comparar elementos e, se necessário, trocá-los de lugar.

A função trocar apresentada no código é uma função auxiliar em linguagem C utilizada para **trocar os valores de duas variáveis inteiras**. Ela recebe como parâmetros dois ponteiros (int* a e int* b), que apontam para os endereços das variáveis a serem trocadas. Usando uma variável temporária (temp), a função armazena, por um período, o valor de *a, atribui o valor de *b a *a, e, por fim, coloca o valor original de *a (armazenado em temp) em *b. Essa função é bastante utilizada em **algoritmos de ordenação**, como o bubble sort, para reorganizar os elementos de um vetor de forma eficiente e organizada.

```c
void trocar(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

A função de troca é usada em diversos algoritmos para reorganizar os elementos conforme o critério definido.

Conhecer diferentes métodos de ordenação permite escolher a melhor estratégia para organizar dados de forma eficiente. Dominar os métodos simples e os mais sofisticados é importante para desenvolver algoritmos robustos e para compreender o funcionamento interno de muitas operações utilizadas em software profissionais e bibliotecas padrão.

Método de ordenação bubble sort

O vídeo a seguir apresenta o funcionamento do bubble sort, um algoritmo de ordenação simples e didático que utiliza comparações e trocas sucessivas para organizar vetores, introduzindo conceitos fundamentais de ordenação e eficiência.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Também conhecido como ordenação por bolha, bubble sort é um dos algoritmos de ordenação mais conhecidos e utilizados para fins didáticos. Embora não seja o mais eficiente em termos de desempenho, é excelente para ensinar os fundamentos da ordenação e o uso de comparações e trocas em estruturas de dados. Seu funcionamento é baseado em comparar pares de elementos adjacentes e trocá-los de lugar caso estejam na ordem errada, como se os maiores valores “borbulhassem” até o final do vetor a cada passagem.

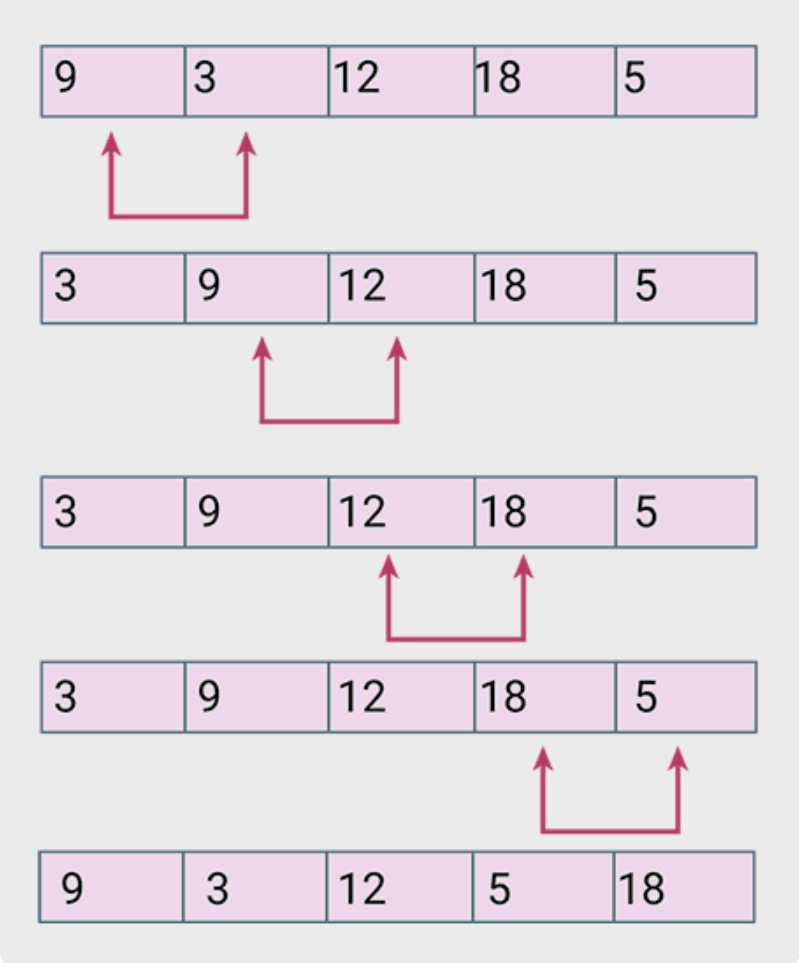
Como funciona o bubble sort?

O algoritmo percorre várias vezes o vetor de elementos. Em cada passagem, ele compara dois elementos vizinhos. Assim, se o da esquerda for maior que o da direita, eles são trocados. O maior valor vai "subindo" até o fim da lista, como uma bolha.

Esse processo se repete até que nenhum elemento precise ser trocado, o que significa que o vetor está ordenado, conforme a imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

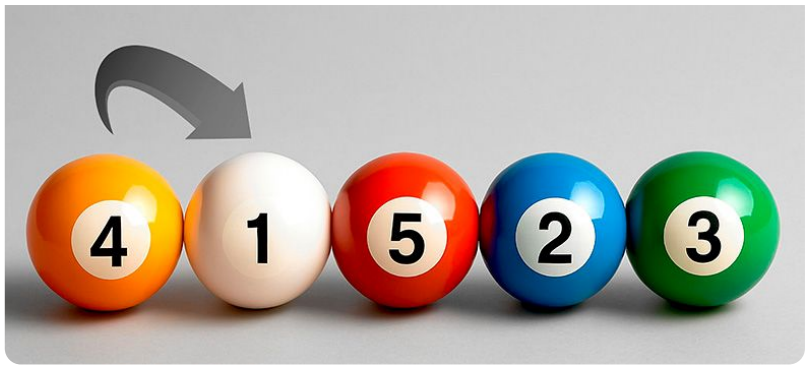


Bubble sort.

Imagine que você tem um conjunto de bolas numeradas e deseja organizá-las do menor para o maior número. Você começa da esquerda, compara cada par de bolas e troca de lugar quando a da esquerda tem valor maior. A cada rodada completa, a maior bola já está em sua posição correta. Repita isso até que mais nenhuma troca seja necessária — e *voilà*, tudo está em ordem. Confira esse exemplo na imagem adiante.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Conjunto de bolas numeradas.

Exemplo prático em C

Neste caso de implementação do bubble sort, a função trocar é utilizada para **inverter a posição de dois elementos** do vetor sempre que estiverem fora de ordem. A variável i controla o número de passagens pelo vetor, e seu limite é ajustado a cada iteração, pois os maiores valores já se posicionam corretamente ao final da lista. Já a condição $\text{vetor}[j] > \text{vetor}[j+1]$ é a responsável por verificar se os elementos estão na **ordem desejada** (crescente). Se não, realiza-se a troca para garantir a ordenação progressiva a cada ciclo do algoritmo.

```c
c

void bubbleSort(int vetor[], int tamanho) {
    for (int i = 0; i < tamanho - 1; i++) {
        for (int j = 0; j < tamanho - 1 - i; j++) {
            if (vetor[j] > vetor[j + 1]) {
                trocar(&vetor[j], &vetor[j + 1]);
            }
        }
    }
}
```

Neste exemplo, entendemos que:

- A função trocar é usada para inverter dois elementos.
- A variável i controla o número de passagens.
- A comparação $\text{vetor}[j] > \text{vetor}[j+1]$ garante a ordem crescente.

Comparação entre algoritmos básicos de ordenação

A seguir, veja uma breve descrição das características principais de cada um desses algoritmos simples, destacando seus pontos fortes e suas limitações.

Bubble sort

Facilita a implementação, mas faz muitas trocas desnecessárias.

Insertion sort
É ideal para listas quase ordenadas.

Selection sort
Faz menos trocas, mas muitas comparações.

O bubble sort é um ótimo ponto de partida para entender algoritmos de ordenação. Ele oferece uma base sólida para conceitos como laços aninhados, comparações e trocas. No entanto, seu uso em aplicações reais é limitado devido à sua baixa eficiência.

Método de ordenação insertion sort

Confira, no vídeo a seguir, o funcionamento do insertion sort, um algoritmo de ordenação simples e didático que utiliza comparações e trocas sucessivas para organizar vetores, introduzindo conceitos fundamentais de ordenação e eficiência.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Um dos algoritmos mais populares para introduzir estudantes ao conceito de ordenação é o insertion sort, ou ordenação por inserção. Embora não seja o mais eficiente para grandes volumes de dados, ele é bastante intuitivo e se destaca quando aplicado a listas pequenas ou quase ordenadas. Sua lógica se assemelha à forma como muitas pessoas naturalmente organizam cartas na mão: insere-se cada nova carta no lugar correto, mantendo as anteriores organizadas.

Como funciona o insertion sort?

O algoritmo percorre o vetor da esquerda para a direita. A cada passo, ele seleciona um elemento e o compara com os anteriores, movendo-o para a direita até encontrar a posição correta em que o valor deve ser inserido. Assim, o vetor é construído de maneira ordenada, um elemento por vez. Veja o passo a passo!

1. O primeiro elemento é considerado já ordenado.
2. O algoritmo verifica onde o segundo elemento se encaixa entre os anteriores.
3. Os valores maiores que o atual são deslocados para a direita para abrir espaço.

Esse processo se repete até que todos os elementos estejam na posição correta.

Imagine que você está organizando cartas numeradas em sua mão. Você pega uma carta de cada vez do baralho e a coloca no lugar certo em relação às que já estão na sua mão. Se a carta nova for menor que a última, você move as outras até encontrar a posição ideal — e insere a nova carta ali. Ao final, suas cartas estão todas em ordem crescente, como na imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

Insertion sort com cartas.

Esse comportamento é o que o insertion sort faz com os elementos de um vetor.

Exemplo prático em C

Veja uma implementação do algoritmo insertion sort em linguagem C. O vetor é percorrido a partir do segundo elemento, e cada valor é comparado aos anteriores para ser inserido no local correto. O deslocamento dos valores maiores garante a ordenação crescente do vetor. Confira!

```c
void insertionSort(int vetor[], int tamanho) {
    for (int i = 1; i < tamanho; i++) {
        int chave = vetor[i];
        int j = i - 1;

        // Move os elementos maiores que a chave uma posição à frente
        while (j >= 0 && vetor[j] > chave) {
            vetor[j + 1] = vetor[j];
            j--;
        }

        vetor[j + 1] = chave;
    }
}
```

No exemplo, vemos que:

- A variável chave armazena o valor que está sendo inserido.

- O laço while desloca os valores maiores que chave.
- Quando a posição correta é encontrada, o valor é inserido com $vetor\ [j+1] = chave$.

Considerações finais sobre o insertion sort

Trata-se de um algoritmo simples, eficiente em situações específicas e excelente para entender o conceito de inserção ordenada. Ele é bastante utilizado como ponto de partida para estudar algoritmos mais avançados, e seu comportamento é muito vantajoso quando o vetor já está parcialmente em ordem.

Embora não seja recomendado para grandes volumes de dados, sua clareza e facilidade de implementação tornam o insertion sort uma ferramenta valiosa para qualquer programador iniciante.

Método de ordenação selection sort

Acompanhe o próximo vídeo, que apresenta o funcionamento do selection sort, introduzindo conceitos fundamentais de ordenação e eficiência.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

O selection sort, ou ordenação por seleção, é outro algoritmo clássico bastante utilizado no ensino de estruturas de dados. Assim como o bubble sort, ele é simples de entender e fácil de implementar, o que o torna ideal para fins educacionais. Sua principal característica é a busca do menor (ou maior) elemento da lista a cada iteração, realizando trocas apenas ao final de cada passagem — o que reduz a quantidade de trocas em comparação a outros métodos simples.

Como funciona o selection sort?

O algoritmo percorre o vetor e, a cada passo:

- Encontra o menor valor entre os elementos restantes.
- Troca esse valor com o primeiro elemento da parte ainda não ordenada.

Esse processo é repetido, fazendo com que os menores valores sejam gradativamente colocados no início do vetor até que toda a estrutura esteja ordenada. Assim:

- A ordenação ocorre do início ao fim do vetor.
- A cada iteração, a menor (ou maior) chave restante é posicionada de modo certo.
- O número de trocas é reduzido em relação ao bubble sort, pois há apenas uma troca por passagem.

Imagine que você está organizando livros por tamanho em uma estante. Você olha todos, escolhe o menor e o coloca no início. Em seguida, repete o processo com os livros restantes, sempre selecionando o menor entre os que ainda não foram organizados. Ao final, todos os livros estarão ordenados do menor para o maior, como mostra a imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

Selection sort com livros.

Qual é a ideia por trás do selection sort? Selecionar o menor e colocá-lo na posição correta, passo a passo.

Exemplo prático em C

O algoritmo utiliza duas variáveis principais: uma para percorrer o vetor e outra para armazenar a posição do menor elemento encontrado em cada passagem. Veja!

```c
void trocar(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void selectionSort(int vetor[], int tamanho) {
    for (int i = 0; i < tamanho - 1; i++) {
        int indiceMenor = i;
        for (int j = i + 1; j < tamanho; j++) {
            if (vetor[j] < vetor[indiceMenor]) {
                indiceMenor = j;
            }
        }
        if (indiceMenor != i) {
            trocar(&vetor[i], &vetor[indiceMenor]);
        }
    }
}
```

Observe que, no exemplo:

- indiceMenor armazena a posição do menor valor encontrado.
- O segundo laço busca o menor valor a partir da posição atual.
- Após a busca, ocorre apenas uma troca por iteração.

Considerações finais sobre o selection sort

Podemos afirmar que se trata de um algoritmo didático e útil para compreender os conceitos de seleção e posicionamento progressivo de elementos. Apesar de sua **eficiência limitada em grandes volumes de dados** devido à sua complexidade, ele tem **vantagem no número reduzido de trocas**, o que pode ser um diferencial em contextos específicos.

Porém, por sua clareza e lógica simples, o selection sort continua sendo um excelente ponto de partida no estudo de algoritmos de ordenação, preparando o terreno para o entendimento de técnicas mais avançadas.

Hora de codar

Veja, no próximo vídeo, como implementar e comparar de métodos simples de ordenação, analisando seu comportamento em diferentes cenários e avaliando o número de comparações e trocas em listas ordenadas, inversas e aleatórias.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Você é responsável por desenvolver um módulo de ordenação para um sistema educacional que precisa organizar listas de dados pequenos e médios, como notas de alunos, tempos em corridas escolares ou nomes por ordem alfabética. O desafio é escolher qual método de ordenação simples oferece melhor desempenho e clareza para ser utilizado nesse contexto.

Antes de tomar a decisão final, você deverá implementar três algoritmos clássicos de ordenação:

- **Bubble sort**
- **Insertion sort**
- **Selection sort**

Além disso, você também deverá avaliar o comportamento de cada um em listas com diferentes padrões de entrada:

- Lista já ordenada (melhor caso).
- Lista em ordem inversa (pior caso).
- Lista em ordem aleatória (caso médio).

Desafio: mestre

O vídeo a seguir propõe a implementação de diferentes algoritmos de ordenação, a análise de seu desempenho, a aplicação de uma busca binária otimizada e a possibilidade de o jogador decidir qual estratégia utilizar com base nas condições do jogo. Imperdível!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O que você vai fazer?

Prepare-se para o desafio final! Imagine que você chegou ao momento decisivo do jogo: a **última safe zone** está se fechando e apenas os jogadores mais habilidosos sobreviveram até aqui. Para escapar da ilha, o jogador precisa montar **um plano estratégico de fuga** construindo uma torre de resgate com os **componentes certos, na ordem exata e com a prioridade certa**. Assim, neste nível, você é responsável por desenvolver o **módulo avançado de organização dos componentes** necessários para a missão final.

Sua tarefa é implementar **diferentes algoritmos de ordenação**, analisar seu desempenho, aplicar uma **busca binária otimizada** e permitir ao jogador decidir qual estratégia usar com base nas condições de jogo. Esta fase integra e consolida os conhecimentos de ordenação, busca eficiente e modularização.

Você deve criar um sistema de priorização e montagem de componentes da torre de fuga, em que o jogador pode **escolher diferentes critérios de ordenação** (por nome, tipo ou prioridade) e, após ordenar, realizar **busca binária** por um item-chave que destrava a ativação da torre. Além disso, o sistema deve **mostrar o número de comparações** para que o jogador entenda os impactos da escolha do algoritmo.

Requisitos funcionais

Confira as principais funcionalidades do sistema a serem implementados:

1. **Criação de structs**:

    - Componente: com os campos char nome[30], char tipo[20], int prioridade.

2. **Entrada dos dados**:

    - Cadastro de até 20 componentes necessários para a montagem da torre.
    - Os dados devem incluir o nome do componente (ex: “chip central”), seu tipo (ex: “controle”, “suporte”, “propulsão”) e a prioridade (de 1 a 10).

3. **Opções de ordenação**:

    - **Bubble sort:** ordenar por nome (string).
    - **Insertion sort:** ordenar por tipo (string).
    - **Selection sort:** ordenar por prioridade (int).

4. **Busca binária:**

    - Aplicável apenas após a ordenação por nome.
    - Deve localizar o **componente-chave** para iniciar a montagem.

5. **Medição de desempenho:**

    - Contar número de comparações realizadas em cada ordenação.
    - Mostrar o tempo de execução de cada algoritmo (usar clock()).

6. **Montagem final**:

    - Exibir todos os componentes ordenados conforme a estratégia escolhida.
    - Confirmar visualmente a presença do componente-chave.

Requisitos não funcionais

Considere também os seguintes critérios relevantes durante o desenvolvimento:

1. **Interface amigável**: um menu interativo deve ter essa característica.

2. **Desempenho educacional**: o programa deve fornecer feedback numérico sobre comparações e tempo.

3. **Clareza no código**: todas as funções e blocos principais devem estar comentados.

4. **Modularização adequada**: cada algoritmo deve estar em uma função separada, facilitando comparação e manutenção.

Instruções detalhadas

A seguir, veja os elementos básicos que devem compor a estrutura do programa:

1. **Bibliotecas necessárias:** stdio.h, stdlib.h, string.h, time.h.

2. **Funções obrigatórias**:

- selectionSortPrioridade(Componente[], int)
- insertionSortTipo(Componente[], int)
- selectionSortPrioridade(Componente[], int))
- buscaBinariaPorNome(Componente[], int, char[])
- mostrarComponentes(Componente[], int)
- medirTempo(void (*algoritmo)(), ...)

3. **Entrada e saída**:

- Use fgets para capturar strings com segurança.
- Exibir o vetor de componentes formatado com nome, tipo e prioridade após cada operação.

Comentários adicionais

Com esse desafio, você consolidará a sua compreensão sobre estratégias clássicas de ordenação, implementação de busca binária e análise de desempenho em tempo real. Ao utilizar algoritmos didáticos, como bubble sort, insertion sort e selection sort, você desenvolve intuição sobre a eficiência de diferentes métodos e entende os efeitos da ordenação no comportamento de algoritmos de busca.

Entregando seu projeto

Finalize sua entrega seguindo os passos a seguir para que tudo esteja corretamente versionado e disponível no GitHub:

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, a fim de 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

Acesse o 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 link que você deve enviar no SAVA.

Explore a estrutura do ambiente do GitHub

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 que contém 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, 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!

Você chegou ao final de uma jornada desafiadora, estratégica e repleta de aprendizados essenciais para sua formação como programador. Ao longo dos três níveis — novato, aventureiro e mestre — você não apenas simulou a experiência de sobrevivência em uma ilha inspirada no universo de Free Fire, como também desenvolveu, testou e analisou soluções com base em estruturas de dados, algoritmos de ordenação e busca, modularização e desempenho computacional.

Agora que você domina esses conceitos, aqui vão algumas dicas para continuar evoluindo:

- **Pratique sempre:** programação é como um jogo — quanto mais você joga, mais habilidade e precisão desenvolve.
- **Melhore o que criou aqui:** adicione menus gráficos, salve dados em arquivos, crie modos multiplayer ou tempo real.
- **Refatore e documente seu código:** treine sua clareza lógica e se prepare para trabalhar em equipe.
- **Resolva problemas reais:** as estruturas que você usou aqui são aplicadas em jogos, apps de entrega, redes sociais, sistemas bancários e muito mais.
- **Continue codando, estudando e evoluindo:** o mercado de tecnologia precisa de pessoas criativas, organizadas e apaixonadas por resolver desafios.

Você mostrou que é capaz de sobreviver com lógica, vencer com estratégia e programar com qualidade.

Missão cumprida, sobrevivente!

Nos vemos no próximo desafio!

Continue praticando

O que transforma conhecimento em habilidade é a prática constante, e nada melhor do que desenvolver novos projetos para consolidar o aprendizado. Veja algumas ideias para continuar evoluindo:

Gestor de senhas pessoais

Implemente um sistema local de gerenciamento de senhas com busca sequencial e ordenação alfabética. Inclua funcionalidades de cadastro, remoção, listagem e verificação por nome do site. Explore também a organização por tipo de serviço (e-mail, redes sociais, bancos etc.).

Ranking de jogadores com estatísticas

Construa um sistema para registrar pontuações e gerar rankings em jogos. Utilize structs, ordenação e busca por nome. Inclua estatísticas como partidas jogadas, vitórias, derrotas e taxa de acerto.

Quiz de lógica com banco de perguntas

Monte um jogo de perguntas e respostas com temas variados (lógica, matemática e história, por exemplo). As perguntas devem ser armazenadas em vetores ou listas. Você pode aplicar ordenação por dificuldade e busca para localizar perguntas por categoria.

Simulador de check-in em aeroporto

Desenvolva um sistema de filas usando listas encadeadas para simular o check-in de passageiros. A cada novo cliente, adicione-o à fila. Além disso, permita atender passageiros, reordenar por prioridade e listar os que aguardam atendimento.

Referências

ASCENCIO, A.; CAMPOS, E. **Fundamentos da programação de computadores**. 3. ed. São Paulo: Pearson Prentice Hall, 2012.

CORMEN, T. *et al.* **Algoritmos**: teoria e prática. 4. ed. Rio de Janeiro: GEN LTC, 2023.

FORBELLONE, A.; EBERSPÄCHER, H. **Lógica de programação**: a construção de algoritmos e estruturas de dados. 3. ed. São Paulo: Pearson Makron Books, 2005.

MANZANO, J.; OLIVEIRA, J. **Algoritmos**: lógica para desenvolvimento de programação de computadores. 27. ed. São Paulo: Érica, 2016.

TANENBAUM, A.; LANGSAM, Y.; AUGENSTEIN, M. **Estruturas de dados usando C**. São Paulo: Pearson Makron Books, 1995.