Listas, pilhas, filas e deques

Prof. Leandro de Mattos Ferreira

Descrição

As estruturas de dados, lista, fila, pilha e deque, bem como suas aplicações, operações e características. A estrutura de lista circular e a sua aplicação em jogos digitais.

Propósito

O conhecimento de estrutura de dados permite ao projetista escolher qual estrutura mais se adequa à solução de um problema ou a um programa. A utilização da estrutura correta pode trazer melhorias de desempenho significativas a um programa.

Objetivos

Módulo 1

Lista em alocação contígua

Reconhecer o conceito de lista em alocação contígua.

Módulo 2

Lista em alocação encadeada

Reconhecer o conceito de lista em alocação encadeada.

Módulo 3

Filas, pilhas e deques

Aplicar as estruturas de filas, pilhas e deques.

Módulo 4

Lista circular e suas aplicações em jogos digitais

Analisar a estrutura de lista circular e suas aplicações.

Introdução

A construção de programas de computador vai muito além de apenas escrever linhas de código para realizar um objetivo. A boa programação se inicia com o entendimento de qual problema se está tentando solucionar e, a partir daí, você deve escolher as ferramentas mais adequadas para chegar a uma solução rápida (eficiente) e correta (efetiva).

As principais ferramentas na solução de um problema por meio de programação são os algoritmos e as estruturas de dados. A escolha do algoritmo e da estrutura de dados corretos pode

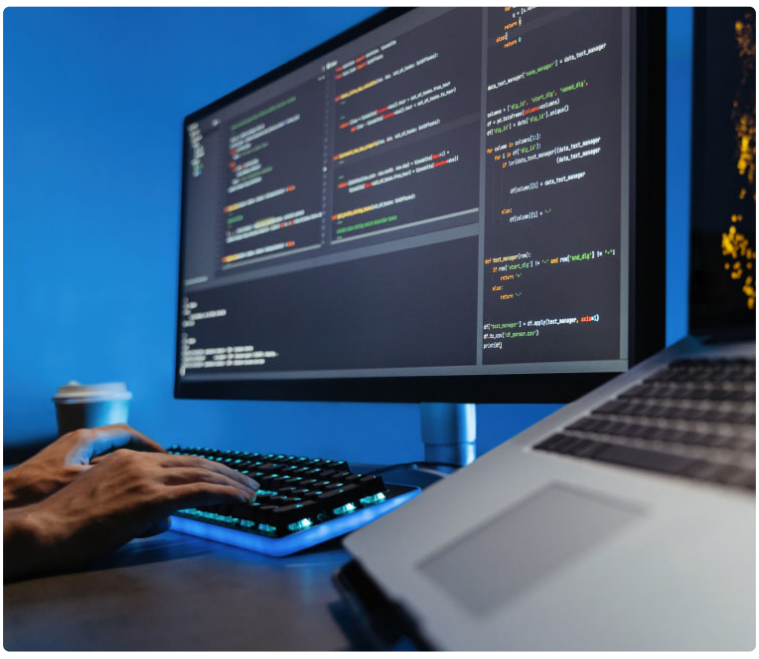
resultar na diferença entre o seu programa resolver o problema rapidamente ou demorar anos para rodar.

Neste conteúdo, você será apresentado ao conceito de estrutura de dados, conhecendo as estruturas: listas, pilhas, filas, deques e listas circulares. Essas estruturas formam a base de conhecimento das estruturas que são normalmente usadas em programas de computador.

Os conceitos básicos para cada estrutura são: a sua funcionalidade; suas operações básicas (busca, inserção e remoção) e suas aplicações. Esses conceitos serão explicados para cada estrutura, e você poderá perceber as diferenças entre elas.

Para completar este assunto, assista ao vídeo e confira as listas circulares e suas aplicações em jogos digitais:

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



1 - Lista em alocação contígua

Ao final deste módulo, você será capaz de reconhecer o conceito de lista em alocação contígua.

Implementação

Confira o conceito de lista e como esta estrutura de dados é representada em memória quando usamos alocação contígua e, ainda, como é a alocação de espaço na lista.

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

Os dados são armazenados em computadores na sua memória principal, também chamada de memória RAM, geralmente em variáveis. Contudo, quando você precisa armazenar e lidar com uma grande quantidade de dados, pode usar uma estrutura de dados de simples operação: a lista em alocação contígua.

O que é lista em alocação contígua?

Uma lista (também chamada de lista linear) é um conjunto de itens organizados sequencialmente (também chamados de nós), que, geralmente, guardam alguma relação entre si. Nessa lista, existe um item inicial $L[0]$ e todos os demais itens $L[i]$ são precedidos por um item $L[i-1]$. Um nó é composto por campos que armazenam informações e um dos campos, geralmente, serve como identificador, sendo comumente chamado de chave.

Dica

A forma mais simples de armazenar uma lista na memória é reservar uma quantidade de espaço contíguo na memória para guardá-la. Nessa estrutura, cada elemento da lista ocupará, sequencialmente, um espaço.

Imagine uma lista de n elementos, na qual cada elemento tem tamanho t. Se o item $L[i-1]$ está no endereço de memória X, então $L[i]$ estará armazenada no endereço $X+t$.

Observe o exemplo a seguir, no qual cada elemento da lista é composto por campos chave e nome:

Índice da lista	Endereço de Memória	chave
2	OxFF750172	28730
1	OxFF750168	42
0	OxFF750160	257

Tabela: Armazenamento de lista em memória.
Leandro de Mattos Ferreira.

Alocando espaço para sua lista

Dependendo da sua linguagem de programação de preferência, você pode ter a opção entre alocar espaço para sua lista de forma estática ou dinâmica.

> ### Mas o que significa cada uma dessas opções?

Em uma alocação **estática**, você reserva um espaço da memória suficiente para guardar o que considera o número máximo de nós de sua lista. Durante o uso da lista, você utiliza uma variável local para guardar o tamanho atual da lista, que pode variar durante a execução do programa.

Exemplo

Você pode alocar o espaço contíguo para uma lista de, no máximo, 1.000 nós. Ao iniciar o programa, a lista possui apenas 10 itens. Você teria, então, uma variável **Max_lista**, com valor 10, que armazena o número atual de nós da lista.

Caso você adicione 4 elementos à lista, você atualizaria a variável **Max_lista** para 14.

Já na alocação **dinâmica** você vai guardando espaço em memória para armazenar sua lista conforme a execução do programa. Normalmente, as linguagens de programação que permitem esse tipo de alocação possuem funções específicas para fazê-lo.

Exemplo

Em C/C++, existem as funções **malloc**() e **free**() para alocar e liberar espaços de memória, respectivamente. Um dos fatores relevantes para

a alocação dinâmica é que o programador fica responsável por alocar e liberar quantidade suficiente de memória para armazenar suas listas.

Operações em lista em alocação contígua

Confira um exemplo de código em Python das operações de busca, inserção e remoção em alocação contígua.

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

Para cada estrutura de dados que você acompanhará neste conteúdo, veremos as três operações básicas que podem ser executadas sobre todas as estruturas: busca, inserção e remoção.

Busca em listas em alocação contígua

No caso de haver uma lista em alocação sequencial, você pode buscar determinado nó, por meio de sua chave, simplesmente percorrendo a lista e procurando por essa chave.

> Imagine que você possui uma lista ***L*** contendo ***n*** nós compostos por ***[chave,nome]***.

Em Python, a lista é um tipo nativo, e você pode buscar por determinado elemento com a função index() já pronta, veja:

Python

Caso queira implementar a busca, o código a seguir apresenta a busca por uma chave k numa lista L com n nós.

Python

Você pode testar a função buscaLista usando o seguinte código::

Python

No caso da lista L estar ordenada por chave (em ordem crescente), você pode melhorar a busca fazendo o laço acabar, caso a chave encontrada seja maior do que a chave buscada. Experimente!

O código a seguir mostra esse caso:

Python

Experimente testar a função buscaOrdenada usando o seguinte código:

Python

Complexidade na busca em lista linear

Você pode analisar a complexidade das duas funções de busca apresentadas? Vamos lá!

Vamos começar com a função **buscaL()**. Podemos assumir que cada comparação de chaves leva um tempo T. A função percorre a lista procurando a chave k até encontrá-la. No seu pior caso, a chave estará na última posição. Nesse caso, a função terá percorrido toda a lista L, executando nT operações. Logo a complexidade de pior caso de **buscaL** é $O(n)$, ou linear.

No caso da função **buscaOrdenada**(), temos a mesma premissa de que cada comparação leva o tempo T. O pior caso é novamente quando o elemento buscado é o último da lista, pois a função percorrerá todos os elementos da lista até encontrá-lo.

Dessa forma, a função terá executado nT operações, tendo sua complexidade de pior caso $O(n)$, ou linear.

> ### Se a complexidade da busca é a mesma em ambas as listas, por que usá-las? Pense sobre isso!

No caso da lista ordenada, a busca é mais eficiente em descobrir quando o elemento buscado **não** está na lista. Nesse caso, a busca para qualquer chave maior que a chave buscada é vista. Ao contrário, na lista não ordenada, você **sempre** tem que percorrer toda a lista para dizer que uma chave não está nela.

Isso pode parecer pouco em listas pequenas, contendo algumas dezenas de nós, mas para listas com milhões de nós, pode fazer muita diferença.

Há um preço a se pagar para manter uma lista ordenada, sabia? Mas depois falamos sobre isso!

Inserção em lista em alocação contígua

A segunda operação básica em uma estrutura de dados é a inserção de um novo nó. No caso da lista sequencial não ordenada, você deve colocar o novo nó após o último elemento da lista.

A lista em Python já possui funções para inserir um nó ao seu final, chamada append(). O código a seguir é um exemplo de seu uso.

Python

Caso queira implementar a inserção por conta própria, o código a seguir exemplifica essa operação. No caso, você possui uma lista L com n nós e quer inserir um novo nó k, veja:

Python

Caso a sua lista seja ordenada, o processo de inserção é mais complicado, pois você tem que buscar a posição correta para inserir o novo nó. Todos os nós posteriores terão que ser “empurrados” uma posição à frente, para abrir espaço. Além disso, caso a chave buscada já exista na lista, não poderemos inserir o nó, o que deve gerar um erro.

O código a seguir mostra essa função:

Python

Que tal testar as funções insereL() insereOrdenada() com o código abaixo!

Python

Complexidade da inserção em lista linear

Você pode analisar a complexidade das duas funções de inserção apresentadas? Vamos lá!

Vamos começar com a função **insereL()**. É uma função bem simples que realiza apenas duas atribuições, não importando o tamanho da entrada. Podemos dizer que sua complexidade de pior caso é $O(1)$, ou constante.

> **Esse é o melhor tipo de complexidade, pois não depende do tamanho de entrada.**

No caso da função **insereOrdenada**(), temos uma função bem mais complexa. Vamos usar a premissa de que cada comparação e cada atribuição leva o tempo T.

Primeiro, a função vai buscar a posição de inserção. Assuma que ela realizou x comparações: o custo foi de xT.

A seguir, os elementos após a posição devem ser "empurrados" para trás, isso necessita de $(n - x)$ atribuições. Ou seja, custa $(n - x)T$.

Somando os dois custos, vemos que a função sempre fará n operações, incorrendo num custo nT. Sua complexidade é $O(n)$ tanto no pior quanto em qualquer caso.

Aqui, você pode perceber o primeiro momento em que a decisão do projetista sobre sua estrutura de dados é muito relevante.

Comentário

Imagine que você tenha uma lista linear com milhões de entradas. Se você faz buscas muito frequentes, mas raramente inserções, pode valer a pena mantê-la ordenada, pois as buscas infrutíferas são resolvidas com muito mais rapidez. Entretanto, as inserções têm custo linear. Se você faz muitas inserções na lista, o custo constante da lista não ordenada pode ser muito mais interessante, e não haveria motivo para mantê-la ordenada.

Esse tipo de análise é o que você deve pensar sobre cada estrutura de dado que irá utilizar, assim como o custo e a frequência de inserções, remoções e buscas, sendo o que embasa fundamentalmente a escolha de uma boa estrutura de dados.

Remoção em lista em alocação contígua

A terceira e última operação básica em uma estrutura de dados é a remoção de um nó. No caso da lista sequencial (ordenada ou não), você deve buscar o nó a ser removido, e depois “puxar” os nós posteriores para preencher o vácuo deixado pelo nó retirado.

A lista em Python possui duas funções prontas para remover elementos de uma lista. A função **remove** (nó k) remove um nó exatamente igual a k, caso exista. A função **pop**(índice i) remove o nó que está na posição de índice i.

Experimente testar essas funções com o código abaixo!

Python

Caso você queira implementar manualmente a remoção, o código a seguir exemplifica essa operação. No caso, você possui uma lista L com n nós e quer remover um nó de chave k :, veja:

Python

Caso a sua lista seja ordenada, o processo de busca pode ser melhorado, permitindo que o erro de chave não encontrada seja detectado logo após ser vista uma chave maior que k. A pequena alteração se dá no laço de busca (linhas 5 6 7), veja:

Python

Você pode testar a função **removeL()** ou sua versão para listas ordenadas usando o seguinte código:

Python

Complexidade da remoção em lista linear

Você pode analisar a complexidade da função de remoção apresentada? Vamos lá!

Vamos começar com a premissa de que uma comparação e uma atribuição levam tempo T. Vejamos:

Primeiro, a função vai buscar a posição de remoção. Assuma que ela realizou x comparações: o custo foi de xT.

→

Segundo, os elementos após a posição devem ser "puxados" para frente, isso necessita de $(n-1-x)$ atribuições. Ou seja, custa $(n-1-x)T$.

Somando os dois custos, vemos que a função sempre fará n operações, incorrendo em um custo $(n-1)T$. Sua complexidade é $O(n)$, tanto no pior quanto em qualquer caso.

Dica

Apenas no caso de remoção infrutífera (o nó buscado não existe na lista) a pequena alteração para a lista ordenada terá uma complexidade melhor, pois você interrompe a busca assim que descobre uma chave maior que a do nó buscado.

Aplicação

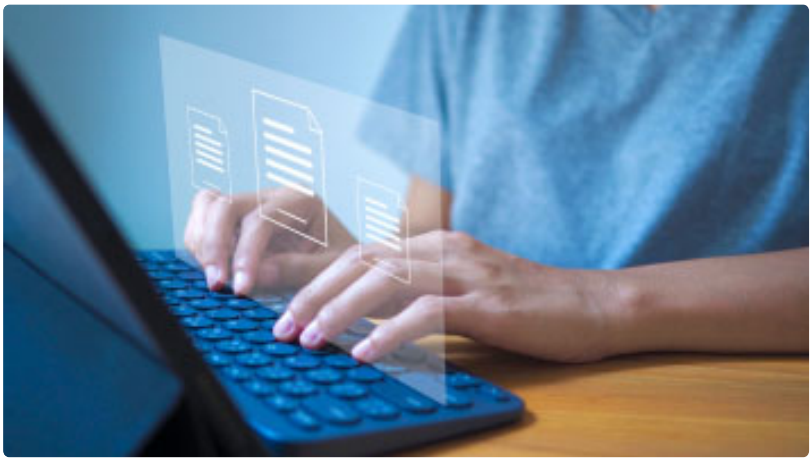
Confira a aplicação do conceito de lista em alocação contígua, para implementar as principais operações (busca, inserção e remoção) de dados cadastrais em memória.

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

Agora que você já conhece a lista em alocação contígua, veja como podemos aplicar essa estrutura de dados e como fazer isso.

Criando uma lista cadastral

Nossa primeira aplicação para uma lista em alocação contígua é uma lista cadastral. Imagine que você tem um mercado e quer organizar uma lista dos seus produtos. O primeiro passo é criar uma chave para essa lista, que possa ser buscada. Nesse caso, podemos criar um campo “produtoID” para ser a chave da nossa lista.



Como segundo item, você pode colocar o nome do produto em um campo “nomeProduto”. Por fim, você pode criar o campo que guarda o preço do produto. O nome "preco" parece ser bom o bastante.

Atenção!

Como produtos serão adicionados e removidos com frequência, vamos manter a lista não ordenada.

Agora, vamos criar a nossa lista com alguns produtos. Vejamos o resultado desejado:

ProdutoID	NomeProduto	preco
15	Laranja	3,00
2	Palmito	20,00

ProdutoID	NomeProduto	preco
35	feijão	5,00

Tabela: Esquema da lista de produtos.
Leandro de Mattos Ferreira.

Experimente criar essa lista em sua linguagem de programação de preferência e criar as funções de busca, inserção e remoção.

Criando uma lista cadastral ordenada

No primeiro exemplo, criamos uma lista não ordenada.

> ### E se agora quiséssemos manter uma lista de clientes de nossa loja?

Essa lista poderia ser mantida em ordem alfabética, para facilitar a consulta.

Vamos criar a lista com o campo “nomeCliente” como nossa chave. Vamos armazenar também sua data de nascimento e data da sua última compra como campos. Veja o exemplo na tabela:

NomeCliente	DataNascimento	DataUltCompra
ana pera	30/11/1980	02/01/2020
joao silva	22/12/1995	19/01/2021
maria santos	03/04/1970	04/01/2022

Tabela: Esquema da lista de clientes.
Leandro de Mattos Ferreira.

Você pode estar em dúvida sobre como manter uma lista ordenada por nomes, mas a maioria das linguagens de programação tem soluções para comparar *strings* e verificar se são menores ou maiores em ordem alfabética.

Caso queira, você pode criar uma função para converter cada letra em um número, que precisará ocupar dois dígitos, por exemplo, *espaço*, valendo 00, **a** 01, **b** 02 e assim por diante.

Diversas codificações desse tipo existem para nossos caracteres, e a mais comum é chamada de ASCII. Internamente, quando se comparam duas strings, os caracteres estão sendo comparados de acordo com suas representações em determinada codificação.

Dica

Experimente criar essa lista em sua linguagem de programação de preferência e criar as funções de busca, inserção e remoção. Lembre-se de que a lista deve ser mantida ordenada pelo nome do cliente.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Uma das formas de implementar uma lista linear é a alocação contígua. Podemos dizer que os elementos desse tipo de lista estão sempre

A ordenados em memória.

B afastados em memória.

C ausentes em memória.

D sequencialmente em memória.

E perdidos em memória.

Parabéns! A alternativa D está correta.

Como a alocação contígua reserva um espaço sequencial de memória para armazenar uma lista, os seus elementos estarão sempre sequencialmente em memória, ordenados ou não.

Questão 2

Uma lista em alocação contígua contém n nós representando filmes, cada nó com os campos: chave, nome, data_de_lançamento. Essa lista é mantida ordenada pelo campo chave, em ordem crescente. A complexidade de pior caso da inserção de um elemento na lista é

A $O(n)$.

B $O(n^2)$.

C $O(n \log n)$.

D $O(1)$.

E $O(n+T)$.

Parabéns! A alternativa A está correta.

Para que um nó seja inserido, a posição correta deve ser buscada, levando x operações de comparação. Depois, todos os nós seguintes deverão ser empurrados em uma posição para abrir espaço para a inserção. Isso leva $n - x$ atribuições. Portanto, somando, teremos n operações e, assim, a complexidade é $O(n)$.

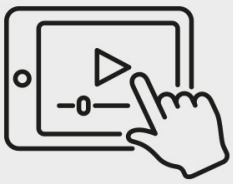
2 - Lista em alocação encadeada

Ao final deste módulo, você será capaz de reconhecer o conceito de lista em alocação encadeada.

Implementação

Confira o conceito de alocação encadeada e como ele é utilizado para representar em memória uma lista e suas duas principais variantes.

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



Você já aprendeu sobre a lista em alocação contígua. Entretanto, viu que, embora de fácil implementação, as operações de inserção e remoção envolvem alterar grande parte da lista, movendo os nós posteriores em uma posição, para frente ou para trás.

Para evitar todo esse trabalho, foram criadas as listas em alocação encadeada.

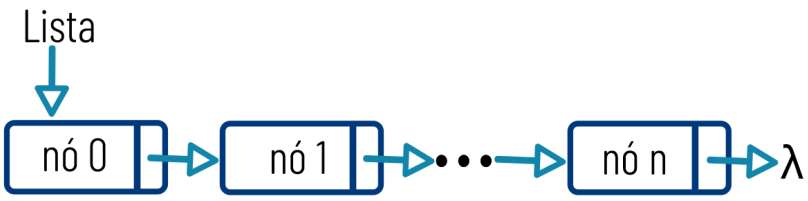
Conceito de alocação encadeada

Em uma lista encadeada, existe o primeiro nó da lista, para onde apontamos a variável da lista (ou seja, a variável **Lista** guarda o endereço deste nó inicial).

Nessa lista, como você já viu, cada nó é composto pelos seus campos, como chave, nome, data etc., mas, além disso, possui um campo especial, um ponteiro para o próximo nó da lista.

> **Essa estrutura permite que os nós estejam salvos em espaços não contíguos da memória, espalhados por diversos endereços distantes entre si, mas que, ainda assim, seja possível percorrer a lista sem se perder.**

Para isso, o ponteiro para o próximo nó deve armazenar o endereço em que está salvo o próximo nó. O último nó da lista, por sua vez, aponta para um valor nulo (representado por lambda λ). Em seguida observe um esquema lógico de lista encadeada:



Exemplificação de lista encandeada.

Em memória, essa lista estará espalhada, com os ponteiros apontando para os endereços dos próximos nós.

	Endereço	Chave
Nó 0	16	157
	...	...
Nó 1	308	158
	...	...
Nó 2	1600	159

Tabela: Representação de lista encadeada na memória.
Leandro de Mattos Ferreira.

Repare que, nesse exemplo, a variável Lista apontará para o endereço 16. A lista conterá 3 nós, contendo chaves ordenadas e sequenciais, mas os endereços de memória não são contíguos.

Alocando espaço para sua lista

No caso da lista encadeada, quando você cria um nó para ser inserido nela, você já está alocando o espaço de memória que irá armazená-lo. Portanto, não precisa se preocupar em alocar previamente espaços para sua lista. Sua única preocupação é que, conforme a lista vai crescendo, o espaço em memória necessário para mantê-la aumenta a ponto de consumir o espaço disponível para seu programa.

Além disso, o tratamento dos ponteiros, apontando para os próximos nós, deve ser feito sempre com cuidado, pois, se você perder o apontamento para o próximo nó, todo o restante da lista estará perdido em endereços desconhecidos da memória.

O código a seguir cria a classe nó e seu construtor, veja:

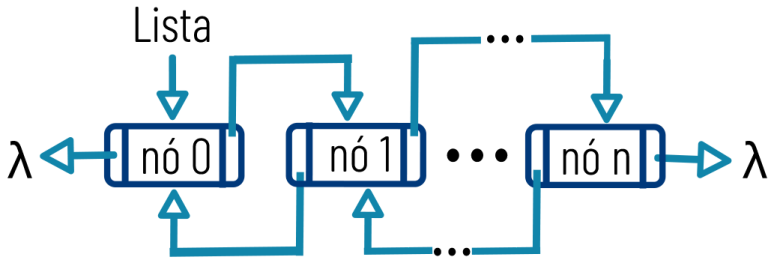
Python

O próximo código cria a classe ListaEncadeada e seu construtor, além de uma função de impressão para facilitar seus testes, veja:

Python

Lista duplamente encadeada

Com o conceito de lista encadeada, você pode percorrer a lista apenas em um sentido, seguindo os ponteiros “próximo”. Existe a estrutura de lista duplamente encadeada, na qual um nó armazena não só o ponteiro para o próximo nó, como também um ponteiro para o nó anterior. Dessa forma, a lista pode ser percorrida em ambos os sentidos, confira:



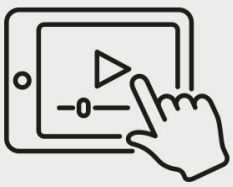
Exemplificação de lista duplamente encandeada.

Entretanto, você consegue apenas começar pelo primeiro nó, pois só tem a variável que aponta para ele. Uma maneira de resolver esse problema é manter uma variável apontando para o último nó da lista, permitindo que você percorra a lista no sentido inverso, seguindo os ponteiros anteriores.

Operações em listas encadeadas

Confira a implementação em Python da busca, inserção e remoção em listas em alocação encadeada.

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



Para a estrutura encadeada, veremos as operações básicas: busca, inserção e remoção. Lembre-se de que o objetivo de utilizar o encadeamento era melhorar a complexidade de inserção e remoção.

Busca em listas em alocação encadeada

No caso de termos uma lista em alocação encadeada, você pode buscar determinado nó, por meio de sua chave, simplesmente percorrendo a lista e procurando por essa chave.

Imagine que você possua uma lista L contendo n nós compostos por [chave, valor, próximo]. O código a seguir apresenta a busca por uma chave k.

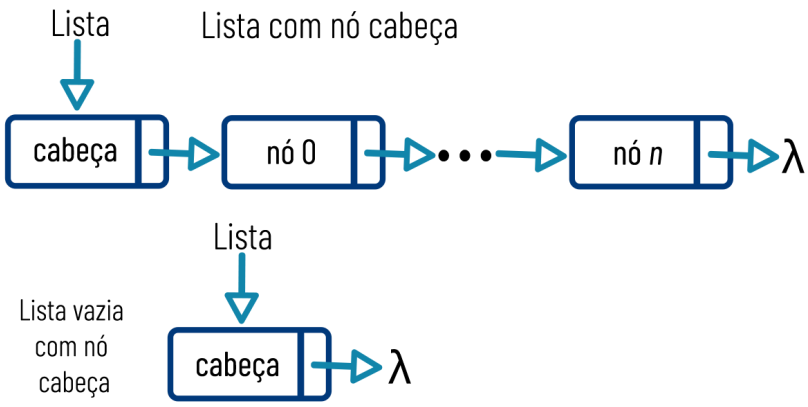
Python

Você percebeu que antes de começar o laço temos que testar com o primeiro nó? Isso acontece porque a lista pode conter apenas um nó. Para evitar esse problema, você pode utilizar um artifício que chamamos de “nó cabeça”.

Atenção!

O princípio do nó cabeça é que a lista sempre terá, ao menos, um nó que não faz parte da lista de verdade, chamado de nó cabeça, que serve apenas para evitar essas checagens extras. Ou seja, se você quiser usar um nó cabeça, ao criar a lista vazia, na verdade, deverá criar contendo um nó (o cabeça) que não armazena valores válidos de um campo.

De mesma forma, uma lista que tenha removido o seu último elemento, ficando só com o nó cabeça, é uma lista vazia. Veja esse artifício:



Exemplificação de lista em alocação encandeada.

Complexidade na busca em lista encadeada

Que tal analisarmos a complexidade da função de busca apresentada? Vamos lá!

Você pode considerar que o laço de busca percorre os elementos da lista em ordem, até encontrar a chave buscada. Perceba que é o mesmo processo da lista em alocação contígua, só que, em vez de percorrer endereços sequenciais da memória, está seguindo os nós por meio dos ponteiros “próximos”.

> Igualmente, no pior caso, a lista será percorrida até o último nó, levando um tempo nT. Portanto, sua complexidade é $O(n)$, ou linear.

No caso da lista estar ordenada, teremos a mesma complexidade, mas a busca infrutífera pode ser interrompida mais cedo, de forma similar ao caso da alocação contígua. Se a chave comparada for maior que a buscada, você já pode parar a busca e retornar à chave não encontrada.

Inserção em lista em alocação encadeada

Aqui, você verá a segunda operação básica: a inserção. Nesse caso, você precisa:

1. Criar um nó a ser inserido, o que vai alocar seu espaço em memória.

2 Apontar o campo próximo desse novo nó para o nó seguinte, de onde irá inserir.

3 Apontar o campo próximo do nó anterior, aonde você vai inserir, para o novo nó.

Primeiramente, vamos considerar o caso da lista não ordenada. Nesse caso, você pode inserir o novo nó ao final da lista, então, no passo 2, apontará para o valor nulo (λ).

Caso você não tenha uma variável apontando para o final da lista, terá que percorrê-la toda até encontrar o último nó. Ou seja, inserir ao final da lista só é eficiente se você tiver uma variável apontando para o final da lista. O código a seguir mostra a inserção no final da lista.

Python

Existe outra solução mais simples, caso você não queira manter uma variável para o final da lista. Você consegue imaginar qual é?

Você pode, em vez de inserir ao final da lista, inserir novos nós no início da lista. Embora pareça estranho, lembre-se de que a lista não é ordenada. Nesse caso, o código é bem simples:

Python

Caso a sua lista seja ordenada, o processo de inserção começa com a busca pela posição correta de inserção, mas, ao encontrá-la, a inserção em si é simples, como as anteriores.

O código a seguir mostra essa função:

Python

Complexidade da inserção em lista encadeada

Você pode analisar a complexidade das três funções de inserção apresentadas? Vamos lá!

Vamos começar com a função **insereFinal**(). Sem uma variável guardando o final da lista, é necessário percorrer toda a lista, sendo assim, tem complexidade $O(n)$. Caso você tenha a variável apontada para o final da Lista, é uma função bem simples que realiza apenas três atribuições, não importando o tamanho da entrada. Podemos dizer que sua complexidade de pior caso é $O(1)$, ou constante.

> ### Esse é o melhor tipo de complexidade, pois não depende do tamanho de entrada.

A função **insereInicio**() também executa apenas duas atribuições, independentemente do tamanho de entrada. Sua complexidade de pior caso é $O(1)$, ou constante.

No caso da função **insereOrdenada**() temos uma função bem mais complexa. Vamos usar a premissa de que cada comparação e cada atribuição leva tempo T.

Primeiro, a função vai buscar a posição de inserção. Assuma que ela realizou x comparações: o custo foi de xT.

A seguir, temos apenas o uso de duas atribuições, com custo constante $2T$. Podemos perceber que, no pior caso, a busca percorre toda a lista, com $x = n$. Nesse caso, no qual a complexidade é $(n + 2)T$, podemos dizer que a complexidade de pior caso é $O(n)$.

Você pode perceber que a complexidade de pior caso não melhorou em relação à alocação contígua, pois já era $O(1)$ no caso não ordenado e $O(n)$ no caso ordenado, e assim se manteve.

A diferença é que, no caso ordenado, quando $x < n$, o programa é muito mais eficiente. Ao achar o local da inserção, o problema é resolvido em tempo constante.

Remoção em lista em alocação encadeada

A terceira e última operação básica em uma estrutura de dados é a remoção de um nó. No caso da lista encadeada, você deve buscar o nó a ser removido, e depois basta fazer o nó anterior apontar para o nó posterior ao nó removido.

O próximo código exemplifica essa operação. No caso, você possui uma lista L e quer remover um nó de chave k.

Python

Caso a sua lista seja ordenada, o processo de busca pode ser melhorado, permitindo que o erro de chave não encontrada seja detectado logo após ser vista uma chave maior que k.

Para testar todas as funções aqui propostas, experimente testar o seguinte código:

Python

Complexidade da remoção em lista encadeada

Qual é a complexidade da função de remoção apresentada? Vamos lá!

Vamos começar com a premissa de que uma comparação e uma atribuição levam tempo T.

Primeiro, a função vai buscar a posição de remoção. Assuma que ela realizou x comparações: o custo foi de $3xT$. (para cada comparação, há também duas atribuições, **noAtual** e **noAnterior**).

> Após encontrar o nó a ser removido, a remoção em si custa 1 operação.

Somando os dois custos, vemos que a função fará $(3x + 1)T$ operações. No pior caso, $x = n$, então, sua complexidade de pior caso é $O(n)$, ou linear.

Entretanto, no caso em que $x < n$, o programa terá o custo efetivamente menor, pois não será necessário mexer em toda a lista. No caso de remoção infrutífera (o nó buscado não existe na lista), a pequena alteração para a lista ordenada terá uma complexidade melhor, pois você interrompe a busca assim que descobre uma chave maior que a do nó buscado.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Quando utilizamos uma estrutura encadeada, os nós possuem ponteiros para outros nós. Em uma lista simplesmente encadeada, há apenas uma ligação entre os nós. Para garantir que a lista permaneça funcional ao remover um nó, você deve seguir quais passos?

A Apontar o ponteiro do nó a ser removido para o nó seguinte a ele.

B

Apontar o ponteiro do nó seguinte ao removido para o nó anterior ao removido.

C Apontar o ponteiro do nó anterior a ser removido para o nó removido.

D Apontar o ponteiro do nó a ser removido para o nó anterior a ele.

E Apontar o ponteiro do nó anterior ao removido para o nó seguinte ao removido.

Parabéns! A alternativa E está correta.

A remoção de um nó se dá fazendo com que o ponteiro do nó anterior a ele aponte para outro nó. Para não perder o encadeamento da lista, o ponteiro do nó anterior tem que apontar para o nó seguinte ao nó removido.

Questão 2

Você possui uma lista ordenada, em ordem crescente de chaves, simplesmente encadeada, com cada nó da lista apontando para o seguinte. Nessa configuração, você pode apenas percorrer a lista em ordem crescente das chaves. Para que possa percorrer a lista em ordem decrescente, você pode utilizar as seguintes alterações:

A Usar nó cabeça e manter variável guardando o último nó.

B Apontar todos os ponteiros para o nó cabeça.

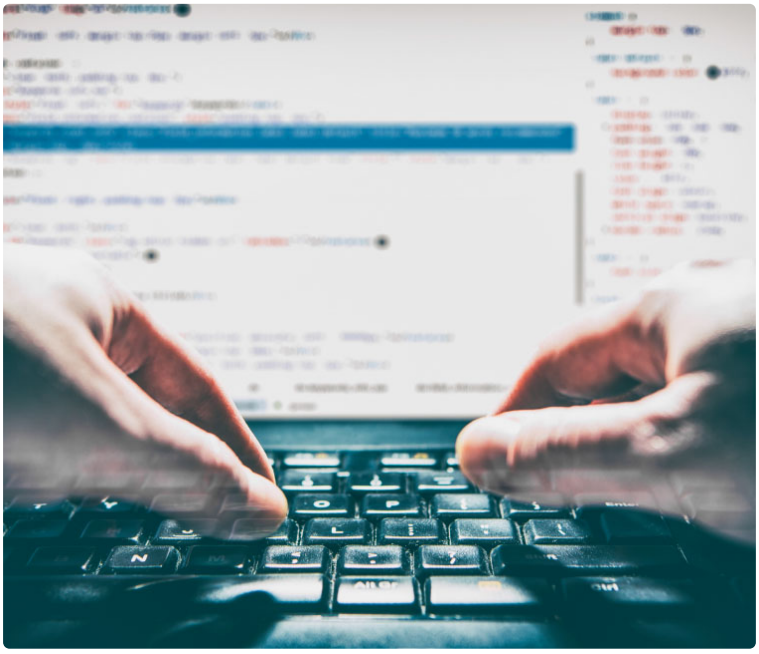
C Inverter todos os ponteiros da lista.

D Fazer o encadeamento duplo e manter variável guardando o último nó.

E Trocar as chaves dos nós par a par, primeiro com último, segundo com penúltimo e assim por diante.

Parabéns! A alternativa D está correta.

Com o uso da variável armazenando o último nó, você consegue iniciar o percurso da lista de trás para frente. Somando a isso o uso de encadeamento duplo, você consegue percorrer a lista de trás para frente, no sentido em que as chaves estarão ordenadas em ordem decrescente.



3 - Filas, pilhas e deques

Ao final deste módulo, você será capaz de aplicar as estruturas de filas, pilhas e deques.

Filas

Entenda agora o conceito de fila, sua política de manutenção, as principais operações e os algoritmos em Pyhton que implementam tais operações.

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

Até o momento, você lidou com estruturas de dados que permitiam inserção e remoção de nós em quaisquer posições. Entretanto, para algumas aplicações, queremos utilizar uma estrutura de dado que remova nós (também chamada de consumir os nós) na mesma ordem em que esses nós foram adicionados. Para isso, usamos a fila.

Conceito de fila

A estrutura de dados fila tem esse nome porque se assemelha conceitualmente às filas do mundo real. Nela, quem chegou primeiro está aguardando mais tempo e também é atendido primeiro. Você pode pensar na fila como uma fila única de um mercado ou banco.

Não importa quantos caixas existam, o primeiro da fila sempre será chamado (aquele que está esperando há mais tempo). Esse comportamento pode ser descrito como “primeiro a entrar, primeiro a sair” (*First In*, *First Out* - FIFO), comumente usado para descrição das filas.

Implementação de fila

A fila pode ser implementada em alocação contígua ou encadeada, como as listas. De fato, a fila é uma lista especial na qual todos os nós são inseridos ao final da lista, e todos os nós são removidos apenas no início da lista.

Para uma alocação encadeada, você pode manter duas variáveis, uma para o início da fila, e outra para o final da fila. Se a fila estiver vazia, ambas as variáveis apontarão para o valor nulo. Se houver apenas um elemento, ambas apontarão para esse elemento. Com mais elementos,

InícioFila apontará para o nó que pode ser removido, e **FinalFila** apontará para o último elemento a entrar, que apontará para o próximo nó a ser inserido. O código a seguir representa a criação da classe e seu construtor.

Python

Para uma alocação contígua, você tem duas opções:

1

Manter sempre o início da fila no endereço inicial da memória alocada, e ir inserindo ao final da fila, no próximo endereço vazio. Para facilitar esse processo, você pode manter uma variável guardando o final da fila. Entretanto, esse método tem a desvantagem de que, quando um nó é removido (do início), todo o restante da fila tem de ser ajustado para frente em memória.

2

Ir alocando os nós sequencialmente em memória, e manter uma variável para o início da fila, e uma para o final da fila. Ao remover um nó (do início), você apenas move a variável **InícioFila** um nó para frente. Ao inserir um novo nó, você move a variável **FinalFila** um nó para frente, para o novo nó. Esta é a opção mais comum em se tratando de alocação contígua.

Quando atingir o final do espaço reservado em memória, você recomeça do início. Se em qualquer momento, ao tentar inserir um nó, o **FinalFila** fosse se mover e alcançar **InicioFila**, sua fila estará cheia e o novo nó não poderá ser inserido. O código a seguir inicializa uma fila com máximo de 10 valores, veja:

Python

Operações em fila

Agora, você verá as operações de remoção e inserção em fila. A operação de busca é igual à de qualquer lista.

Inserção em fila

A inserção em fila deve ocorrer sempre no final da fila.

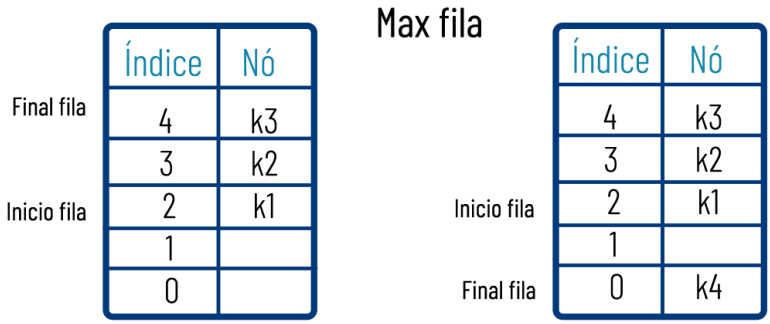
Para o caso encadeado, você pode usar o seguinte código (novoNo deve estar alocado e com próximo apontando para o nulo):

Python

Para o caso contíguo, você possui 4 variáveis: **Fila** guarda o espaço de memória. **MaxFila** guarda o número máximo de elementos na fila, **InicioFila** e **FinalFila** guardam os índices de início e final da fila, respectivamente. Você pode usar o seguinte código:

Python

O detalhe nesse último código fica pelo uso da lógica modular representada pelo operador %. Ao somar 1 na variável **FinalFila**, se você atingir o valor de **MaxFila**, ao aplicar o módulo **MaxFila**, a variável **FinalFila** volta para 0. Veja essa lógica no esquema gráfico das filas a seguir (antes e após a inserção do nó k4):



Esquema de uma fila em Python.

Antes da inserção, o valor de **FinalFila** é 4. **MaxFila** é 5, pois podemos manter até 5 nós na fila. Ao tentarmos inserir k4, o valor de **finalFila** é aumentado em 1, indo pra 5, mas, ao aplicarmos o módulo **MaxFila** (que é 5), **FinalFila** volta para 0. E assim, recomeçamos a inserir nós na parte inicial do espaço alocado em memória.

Complexidade da inserção na fila

Analisando as funções de inserção na fila, e desconsiderando os tratamentos de erro de fila cheia, você pode verificar que a inserção consiste apenas em 2 ou 3 operações de atribuição, portanto, podemos dizer que a inserção em filas é $O(1)$, ou constante.

Remoção da fila

A remoção da fila (às vezes chamado de consumir um nó) deve ocorrer sempre no início da fila. Para o caso encadeado, você pode usar este código:

Python

Para o caso contíguo, você possui 4 variáveis: **Fila** guarda o espaço de memória, **MaxFila** guarda o número máximo de elementos na fila, **InicioFila** e **FinalFila** guardam os índices de início e final da fila, respectivamente. Experimente usar este código:

Python

Agora, para testar a implementação da sua fila encadeada, que tal usar um código como este aqui!

Python

Já para testar sua fila em alocação contígua, você pode usar o seguinte código:

Python

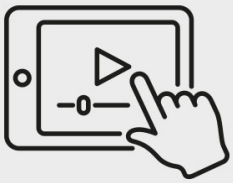
Complexidade da remoção da fila

Analisando as funções de remoção da fila, e desconsiderando os tratamentos de erro de fila vazia, você pode verificar que a remoção consiste apenas em 2 operações de atribuição e, possivelmente, 2 comparações, portanto, podemos dizer que a remoção em filas também é $O(1)$, ou constante.

Pilhas

Confira o conceito de pilha, sua política de manutenção, as principais operações e os algoritmos em Pyhton que implementam tais operações.

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



Após aprender sobre as filas, você, agora, verá a estrutura de dados que permite colocar e remover nós apenas no início da lista: a pilha.

Conceito de pilha

A estrutura de dados pilha tem esse nome porque se assemelha conceitualmente às pilhas (de roupas, caixas, livros etc.) do mundo real. Nela, quem chegou primeiro fica embaixo de todos os outros. Ora, para você remover uma caixa que está abaixo de várias outras, você precisa remover primeiro as de cima.

> **Assim é a pilha: você só pode colocar um novo elemento no topo da pilha ou retirar o elemento do topo da pilha, os demais estão "debaixo", então não podem ser movidos.**

Essa estrutura faz com que o nó mais recente seja sempre o retirado da pilha. Essa política também é chamada de ***Last In First Out***, com o acrônimo LIFO.

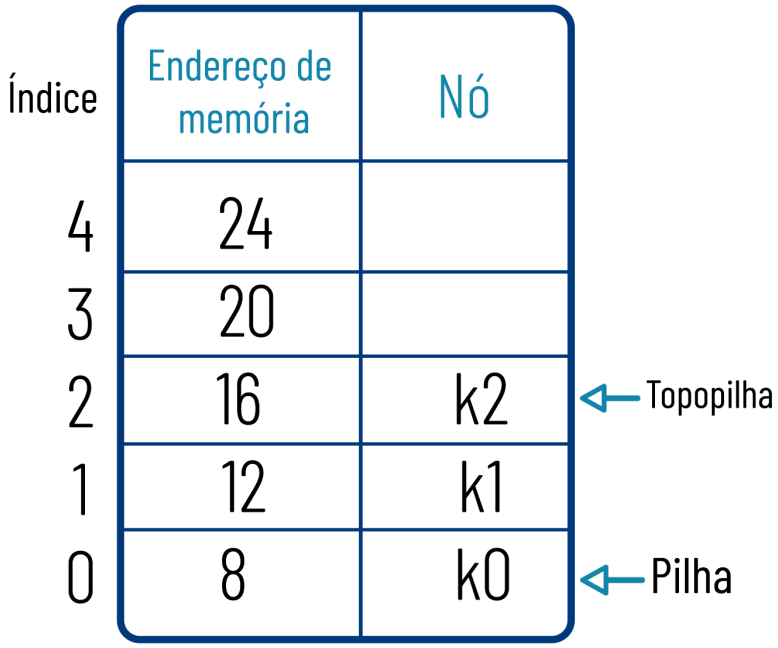
Implementação de pilha

A pilha pode ser implementada em alocação contígua ou encadeada, como as listas. De fato, a pilha é uma lista especial na qual todos os nós são inseridos e removidos apenas no início da lista.

Para uma alocação encadeada, você só precisa manter um ponteiro para o início (comumente chamado de topo) da pilha. Se a pilha estiver vazia, o **TopoPilha** apontará para o valor nulo. O código a seguir apresenta a classe Pilha e seu construtor.

Python

Para uma alocação contígua, você usará o espaço alocado de memória exatamente como uma pilha, enchendo da base (índice 0) para cima, e manterá uma variável **TopoPilha** que guarda o índice do topo da pilha. Deve apenas tomar cuidado para não armazenar mais nós do que o espaço reservado (que pode ser salvo em **MaxPilha**). Veja um exemplo de pilha com essa alocação:



Esquema de uma pilha em Python.

Nessa esquema, podemos ver que a variável Pilha aponta para o início do espaço alocado em memória. **Pilha**[0] contém o nó $k0$, e assim por diante. O topo da pilha está apontado por **TopoPilha**, que tem valor 2. O valor de **MaxPilha** é 5, ou seja, se topo pilha apontar para 4 e tentarmos inserir um nó, deve ocorrer um erro e a inserção não vai acontecer.

Veja o código para inicializar uma pilha nesse formato:

Python

Operações em pilhas

Agora, você verá as operações de remoção e inserção em pilhas. A operação de busca é igual a de qualquer lista.

Inserção em pilha

A inserção na pilha deve ocorrer sempre no topo da pilha. Essa operação é comumente chamada de PUSH.

Para o caso encadeado, você pode usar o código abaixo (novoNo deve estar alocado e com o próximo apontando para o nulo):

Python

Para o caso contíguo, você possui 3 variáveis: **Pilha** guarda o espaço de memória, **MaxPilha** guarda o número máximo de elementos na pilha, **TopoPilha** guarda o índice do topo da pilha. Experimente usar este código:

Python

Complexidade da inserção na pilha

Analisando as funções de inserção na pilha, e desconsiderando os tratamentos de erro de pilha cheia, você pode verificar que a inserção consiste apenas em 2 operações de atribuição, portanto, podemos dizer que a inserção em pilhas é $O(1)$, ou constante.

Remoção da pilha

A remoção da pilha (comumente chamado de POP) deve ocorrer sempre no topo da pilha. Para o caso encadeado, você pode usar este código:

Python

Para o caso contíguo, você possui 3 variáveis: **Pilha** guarda o espaço de memória, **MaxPilha** guarda o número máximo de elementos na pilha, **TopoPilha** guarda o índice do topo da pilha. Experimente este código:

Python

Agora, para testar suas funções de push e pop na pilha, que tal usar o código abaixo?

Python

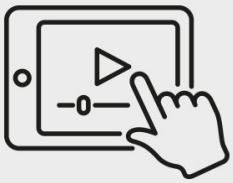
Complexidade da remoção da pilha

Analisando as funções de remoção da pilha, e desconsiderando os tratamentos de erro de pilha vazia, você pode verificar que a remoção consiste apenas em 2 operações de atribuição e, possivelmente, 1 comparação, portanto, podemos dizer que a remoção em pilhas também é $O(1)$, ou constante.

Deques

Confira o conceito de deque, sua política de manutenção, as principais operações e os algoritmos em Pyhton que implementam tais operações.

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



Após aprender sobre as filas e pilhas, você verá a estrutura de dados que permite colocar e remover nós tanto no início como no fim da lista, chamada de deque (*Double Ended QUEue*, ou fila com duas pontas).

Conceito de deque

A estrutura de dados deque é uma generalização da fila e da pilha, essencialmente permitindo que se adicione ou remova nós do início ou do final da lista. Para evitar confusões, as operações são normalmente identificadas com qual lado da fila está sendo alterado. A remoção de um nó do início pode ser chamada de pop_front, enquanto a remoção de um nó ao final pode ser chamada de pop_back.

Implementação de deque

O módulo *collections* do Python já possui uma implementação de deque. Você pode acessá-la usando este comando:

Python

Veja exemplos de operações em deque, como inserção e remoção em ambas as pontas:

Python

Caso queira implementar seu próprio deque, ele pode ser implementado em alocação contígua ou encadeada, como as listas.

Para uma alocação encadeada, você precisa manter um ponteiro para o início e para o final do deque. Se o deque estiver vazio, ambos apontarão para o valor nulo. Usualmente, utilizamos um encadeamento duplo, que permite percorrer a lista em qualquer uma das direções e a partir de qualquer uma das pontas.

Observe os códigos do nó e do deque:

Python

Para uma alocação contígua, você pode usar um vetor circular, como fizemos para a fila. As duas variáveis (**InicioDeque** e **FinalDeque**) indicam as pontas do deque e vão se deslocando conforme os nós são inseridos e removidos. Ao chegar ao fim do espaço alocado e ser incrementada, a variável faz a volta e o índice volta para zero.

De forma similar, se o valor estiver em 0 e for decrementado, passa para o maior valor possível. Normalmente, usamos a lógica modular para fazer esses incrementos/decrementos, assim como vimos na fila. O código a seguir é um exemplo de alocação do deque.

Python

Por fim, quando o espaço do vetor está cheio, temos um deque cheio que não poderá receber novos nós, a menos que o seu espaço seja aumentado dinamicamente.

Operações em deques

Nas operações de remoção e inserção em deques a operação de busca é igual a de qualquer lista.

Inserção em deques

A inserção no deque pode ocorrer no início ou final da estrutura. Quando ocorre no início da estrutura, seu funcionamento é idêntico à inserção na

pilha, apenas ajustando-se as variáveis **InicioDeque** e **FinalDeque**. Essa operação pode ser chamada PUSH_front.

Quando a inserção ocorre no final da estrutura, seu funcionamento é idêntico à inserção na fila. Essa operação também pode ser chamada PUSH_back.

Complexidade da inserção no deque

Considerando que as funções de inserção no deque são essencialmente as mesmas que as da fila e pilha, podemos afirmar que a inserção em deques é $O(1)$, ou constante.

Remoção do deque

A remoção do deque pode ocorrer no início ou no final da estrutura. Quando ocorre no início, seu funcionamento é idêntico à remoção da pilha, apenas ajustando-se os ponteiros de início e final do deque (no caso encadeado) ou as variáveis **InicioDeque** e **FinalDeque**(no caso contíguo). Essa operação pode ser chamada popFront ou popLeft.

Quando a remoção ocorre no final da estrutura, seu funcionamento pode ser chamado popBack ou popRight. Para o caso encadeado, você pode usar o código abaixo:

Python

Para o caso contíguo, você possui 4 variáveis: Deque guarda o espaço de memória, **MaxDeque** guarda o número máximo de elementos do

deque, **InicioDeque** e **FinalDeque** guardam o índice do início e final do deque. Você pode usar este código:

Python

Complexidade da remoção em deques

Analisando as funções de remoção do deque, e considerando que são similares às remoções de pilhas e filas, podemos dizer que a remoção em deques também é $O(1)$, ou constante.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

A estrutura de dados de pilha é muito utilizada para programas de análise sintática. As suas complexidades de inserção, remoção e busca são, respectivamente

A $O(1), O(1)$ e $O(n)$.

B $O(1), O(n \log n)$ e $O(n)$.

C $O(nlogn), O(1)$ e $O(1)$.

D $O(n), O(n)$ e $O(n)$.

E $O(1), O(n)$ e $O(1)$.

Parabéns! A alternativa A está correta.

A inserção só pode ocorrer no topo da pilha e leva tempo constante, ou $O(1)$. Da mesma forma, a remoção só pode ocorrer no topo da pilha, sendo $O(1)$. Já a busca necessita percorrer a pilha e, portanto, no pior caso, leva tempo $O(n)$.

Questão 2

Você está utilizando uma estrutura de dados de fila em seu programa. Em determinado momento da execução, a fila apresenta a seguinte estrutura, representada pelas suas chaves: [5,11,15,20,25,36]. O nó de chave 5 é o início da estrutura. Os próximos três nós a serem consumidos são os nós de chave

A 36, 35, 30.

B 5, 11, 15.

C 5, 15, 25.

D 36, 20, 11.

E 5, 20, 36.

Parabéns! A alternativa B está correta.

A estrutura de fila só permite remoções no seu início. Portanto, as três remoções serão 5, 11 e 15.

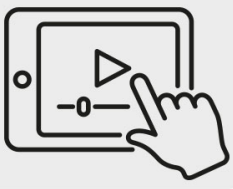
4 - Lista circular e suas aplicações em jogos digitais

Ao final deste módulo, você será capaz de analisar a estrutura de lista circular e suas aplicações.

Lista circular

Confira o conceito de lista circular, sua política de manutenção, as principais operações e os algoritmos em Pyhton que implementam tais operações.

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



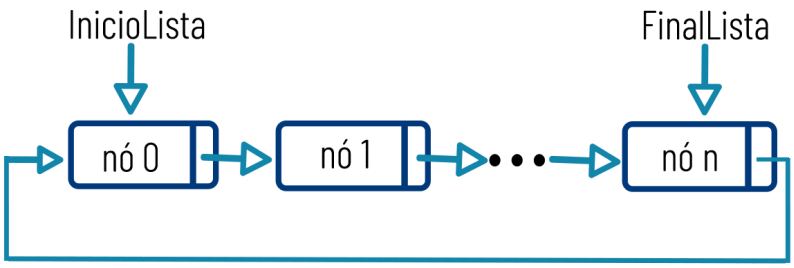
A lista circular é outro tipo de estrutura de dados que não tem fim ou começo. É possível continuar circulando pelos nós indefinidamente. Mas o que exatamente é uma lista circular e como implementá-la? Vamos descobrir!

Implementação da lista circular

A ideia da lista circular é permitir a representação de uma lista de dados cíclica, como um *loop* sem fim. Para isso, você precisa fazer com que o último nó seja seguido pelo primeiro, fechando o laço eterno.

No caso encadeado, isso pode ser facilmente resolvido apontando o ponteiro **próximo** do último nó para o primeiro nó, ao invés de apontar para nulo. Se sua lista for duplamente encadeada, você também precisa apontar o ponteiro **anterior** do primeiro nó para o último.

Entretanto, é importante continuar mantendo duas variáveis, uma apontando para o "início" da lista, e outra, para o "final" da fila. Essas duas variáveis são importantes para você fazer as operações de inserção e remoção da lista. Veja a estrutura:



Exemplificação da implementação da lista curricular.

Para o caso de alocação contígua, você simplesmente terá de considerar que o primeiro endereço (índice 0) é o seguinte ao último endereço (índice NúmeroNos - 1), e vice-versa.

Para fazer isso, você pode usar a matemática modular, com módulo **NumeroNos**, ao incrementar ou decrementar as variáveis **inicioLista** e **FinalLista**.

Operações em lista circular

As operações básicas em listas circulares são inserção, remoção e busca.

Inserção em lista circular

A inserção em lista circular funciona exatamente como na lista linear, com a única diferença que se você inserir após o “final” da lista, você deve atualizar a variável que aponta para o final da lista. No caso de ser encadeada, também precisará atualizar os ponteiros associados. O código a seguir mostra este caso:

Python

Se a lista for ordenada, você deve realizar a busca pelo local de inserção. Tome cuidado apenas quando a inserção for antes do primeiro nó, para atualizar o ponteiro de início da lista e apontar o último nó para o novo nó inicial.

Complexidade da inserção em lista circular

Se a inserção puder ser feita apenas ao final da lista, requer apenas algumas atribuições, tendo complexidade $O(1)$, ou constante.

Caso você use a lista ordenada, a busca pela posição de inserção é $O(n)$, e a colocação dos nós requer apenas passos constantes, logo, a inserção ordenada é $O(n)$.

Remoção em lista circular

A remoção em lista circular funciona de forma semelhante à lista linear, precisando apenas atualizar os ponteiros de início e final da lista, caso um dos **extremos** seja removido. No caso de lista encadeada, experimente usar o seguinte código:

Python

Complexidade da remoção em lista circular

No caso da implementação encadeada, a busca pelo nó a ser removido é $O(n)$, e a remoção do nó requer apenas passos constantes, logo, a remoção é $O(n)$.

No caso de alocação contígua, temos a mesma situação da lista linear. Se a busca levar x etapas, todo o resto da lista precisará ser movido, resultando em $(n - x)$ operações. Dessa forma, a remoção também é $O(n)$ no caso contíguo.

Aplicações da lista circular em jogos digitais

Entenda agora como a lista circular pode ser usada para facilitar a implementação de um jogo digital, mais especificamente, um jogo de cassino.

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

Após aprender sobre a lista circular, você será apresentado a aplicações práticas para o uso de uma lista circular.

Utilizando lista circular em jogos

A lista circular pode ser aplicada em situações que necessitem gerar elementos extraídos **repetidamente** de uma lista limitada, possivelmente seguindo um conjunto especificado de regras. Um exemplo desse tipo de uso são os jogos de cassino.

Exemplo

Você pode implementar uma lista circular de 38 nós que representa uma roleta, com um conjunto não ordenado de nós representando os 38 números da roleta em um arranjo não sequencial. Você, então, pode manter um ponteiro para um elemento da lista que representa a bolinha da roleta. Nesse caso, rodar a roleta seria apenas decidir um número de deslocamento, que seria quantos movimentos você percorre na lista, e o ponteiro apontará para o novo resultado.

Entretanto, o uso mais comum para uma lista circular em jogos digitais é representar um conjunto de jogadores, permitindo que a cada um seja dada sua vez de agir, de forma sequencial, voltando a vez para o primeiro, após o último agir.

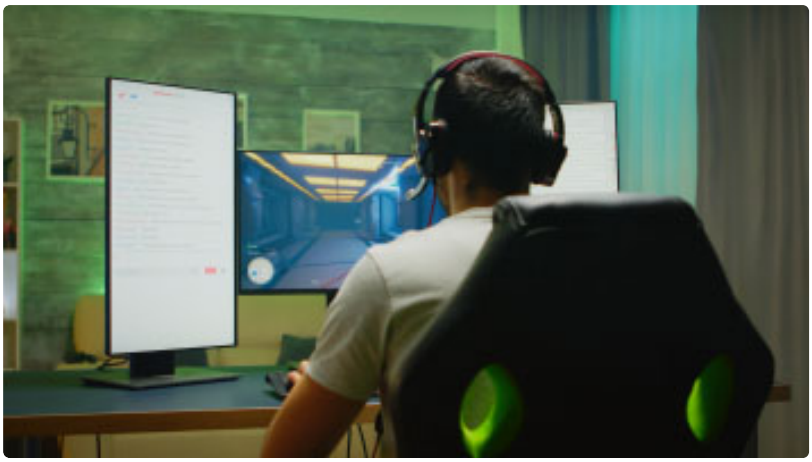
A esse esquema, no qual cada um pode ter uma oportunidade de agir, damos o nome de Round Robin. E listas circulares são formas extremamente práticas de implementar sistemas Round Robin.

Aplicação de fila em alocação contígua em jogos digitais

Uma fila (na qual os nós só podem ser removidos do início e adicionados ao final) em alocação contígua, que ocupa um espaço limitado e fixo, é uma estrutura muito usada em jogos digitais on-line.

Como a velocidade de transmissão de dados pode ser menor que a velocidade de geração de dados, a estrutura citada, no caso, a fila, funciona como um buffer, armazenando os dados até eles serem transmitidos.

A grande vantagem é que o espaço de memória é fixo, e conforme os nós vão sendo consumidos da fila, o espaço vai sendo liberado para novos dados. Essa estrutura pode ser usada para manter os sistemas de conversa (chat) dos jogos, e também para armazenar os comandos introduzidos pelo jogador em seu dispositivo local, até que sejam consumidos pelo servidor.



A complexidade de inserção e remoção é a melhor possível: constante. O espaço de memória é fixo e não precisa ser reajustado e nem cresce com o tempo. Portanto, é uma estrutura ideal para esse tipo de aplicação.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Uma lista circular duplamente encadeada é uma estrutura de dados que possui como principal característica

A busca otimizada.

B remoção lenta.

C ausência de ponteiros nulos.

D inserção em tempo quadrático.

E percurso em apenas um sentido.

Parabéns! A alternativa C está correta.

O fato de a lista ser duplamente encadeada permite que o percurso seja feito em ambos os sentidos: as operações de inserção e remoção são lineares. A característica principal da lista circular é que os ponteiros sempre apontam para outros nós, nunca para ponteiros nulos. O último nó tem seu ponteiro próximo apontando para o nó inicial da lista. O ponteiro anterior do nó inicial, por sua vez, aponta para o último nó.

Questão 2

Em um jogo digital, você quer implementar um elemento de uma máquina caça-níqueis, que mostra um número selecionado de um conjunto fixo de números a cada vez que um botão é apertado. Uma vez definida a ordem desses números, ela não mais se altera, não sendo possível inserir ou remover números, como se estes estivessem dispostos sobre o perímetro de um disco. Para implementar esse elemento, a estrutura ideal é

A uma pilha simplesmente encadeada.

B uma fila duplamente encadeada.

C um deque em alocação contígua.

D uma lista linear.

E uma lista circular.

Parabéns! A alternativa E está correta.

Como a sequência de números é fixa, e não são inseridos ou removidos números, o uso de pilhas, filas ou deques não é justificado, pois sua grande vantagem é a baixa complexidade nas inserções e remoções. Já a lista circular permite que diversos números sejam gerados de forma repetida, sem a necessidade de

recomeçar do início da lista a cada aperto do botão. Portanto, a estrutura ideal é a lista circular.

Considerações finais

Você foi apresentado a diversas estruturas de dados, iniciando pela mais simples: a lista linear. Passamos pela sua implementação, seus usos e principalmente pelas operações básicas de busca, inserção e remoção. Vimos também a complexidade dessas operações, que é o ponto focal na escolha de qual estrutura deve ser usada em cada caso.

Vimos também as estruturas de dados com pequenas variações, que servem para os mais diversos propósitos: as filas, que consomem nós na ordem em que foram inseridos; as pilhas, que inserem e removem nós do topo da estrutura; e os deques, que são uma generalização de filas e pilhas, permitindo operações de inserção e remoção apenas em suas pontas.

Por fim, você aprendeu sobre a lista circular, que permite um percurso contínuo entre seus elementos, e viu a sua aplicação em jogos digitais, principalmente na ordenação dos jogadores em jogos multijogador. Para completar, vimos o importante uso de filas contíguas, também chamadas de buffers de tamanho fixo, para o armazenamento de mensagens e comandos enquanto são transmitidos nos jogos on-line.

Podcast

Para encerrar, ouça sobre os principais conceitos da estrutura de dados lineares.

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

Explore +

Pesquise mais sobre como utilizar as estruturas de dados aprendidas na sua linguagem de programação preferida. Por exemplo, em Python, a documentação de estruturas de dados explica como utilizar cada uma delas. Procure, por exemplo, por “pyhton data structure operations” (o domínio será do tipo docs.python.org).

Referências

SZWARCFITER, J. L.; MARKEZON, L. **Estrutura de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2010.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos teoria e prática**. 3. ed. Rio de Janeiro: LTC, 2012.

FERRARI, R.; RIBEIRO, M. X.; DIAS, R. L.; FALVO, M. **Estruturas de dados com jogos**. 1. ed. Rio de Janeiro: Elsevier, 2014.

Material para download

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

Download material

O que você achou do conteúdo?

Relatar problema