Lista em alocação contíngua

Implementação

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.

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**.

Logo: **L[i] = X + t** ou **L[i] = L[i-1] + 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	nome
2	OxFF750172	28730	João
1	OxFF750168	42	Maria
0	OxFF750160	257	Ana

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.

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.

Operações em lista em alocação contígua

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

```
nomes=['Joao','Maria','Ana']
i=nomes.index('Joao')        #busca o índice do nó com a chave Joao
print(i)
```

Caso queira implementar a busca, o código a seguir apresenta a busca por uma chave **K** numa lista **L** com **N** nós.

```
def buscaLista(k, L, n)
    i=0
    índiceL=-1
    while i<n:
        if L[i]==k:          #nó encontrado
```

```
            indiceL=i    #salva o índice
            i=n+1        #sair do laço
        i=i+1            #segue a procura
    return indiceL

nomes =['Arthur', 'Joao', 'Maria', 'Ana']
i = buscaLista ('Ana', nomes, len(nomes))
print (i)
```

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.

```
def buscaOrdenada(k, L, n):
    i=0
    indiceL=-1
      while (i<n):
            if L[i]>= k:
                if L[i] == k:
                                indiceL=i          #chave encontrada
                                i=n+1              #sair do laço
                else:
                            i=n+1          #chave>k, sair do laço
            else:
              i=i+1                              #continuar busca
    return indiceL                               #-1 indica chave não achada

numeros=[1,2,3,4,6,7,8,9,10]
i=buscaOrdenada(5,numeros,len(numeros))
print (i)
i=buscaOrdenada(3,números,len(números))
print(i)
```

Complexidade na busca em lista linear

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.

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.

```
nomes=['Joao','Maria','Ana']

nomes.append('Arthur')      #insere um novo nó contenho Arthur

print(nomes)
```

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 **n**, veja:

```
def insereL (k, L, n):
    L.append('')    #aumenta um índice na lista
    L [n]=k         #índices na lista iniciam em 0
    n++             #incrementar o número de nós n
```

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.

```
def insereOrdenada (k,L,n)
    i=0                         #início da busca da posição
    posInsercao=-1
    while (i<n):
        if L[i]>= k:
            if L[i] == k:
                return -1       #erro, chave já existe
            else:
                posInsercao =i    #posição achada
                i=n+1           #sair do laço
        else:
            i=i+1               #continuar busca
        if i==n:
            posInsercao=n       #inserção no final da lista
                                #final da busca de posição
    L.append(‘’)                    #aumenta um índice na lista
    i=n                          #inicio do ajuste da lista
    while (i>posInsercao):
        L[i]=L[i-1]             #empurra cada nó para o final
        i=i-1
    L[posInsercao]=k                #insere novo nó
    return posInsercao               #saída normal da função


numeros=[1,2,3,4,6,7,8,9,10]
insereL(12,numeros,len(numeros))
print(numeros)
insereOrdenada(5,numeros,len(numeros))
print(numeros)
```

Complexidade da inserção em lista linear

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.

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.

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.

```
nomes =['Joao', 'Maria', 'Ana', 'Arthur']

nomes.remove('Arthur')

print(nomes)

nomes.pop(2)

print(nomes)
```

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**:

```
def removeL (k,L,n):
    i=0                    #início da busca do nó
    posRemocao=-1
    while (i<n):
        if L[i] == k:
            posRemocao=i   #chave encontrada
```

```
            i=n+1          #sair do laço
        else:
            i=i+1          #continuar busca
    if i==n:
        return -1           #erro, chave não existe
                           #final da busca do nó
    i=posRemocao            #início do ajuste da lista
    while (i<n-1):
        Lista[i]=Lista[i+1]   #puxa cada nó posterior 1 posição
        i=i+1
    L.pop(n-1)              #ajusta o tamanho da lista
    return posRemocao       #saída normal da função
```

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

```
if L[i] == k:
        posRemocao=i       #chave encontrada
        i=n+1              #sair do laço
else:
        if L[i] >k:
            return -1      #erro, chave não existe
        i=i+1
```

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

```
nomes =['Joao', 'Maria', 'Ana', 'Arthur']
i=removeL('Arthur', nomes, len(nomes))
print(nomes)
```

Complexidade da remoção em lista linear

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 **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.

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.

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

ProdutoID	NomeProduto	preco
15	Laranja	3,00
2	Palmito	20,00
35	feijão	5,00

Criando uma lista cadastral 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

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.