**FACULDADE DE COMPUTAÇÃO E INFORMÁTICA**

**CIÊNCIA DA COMPUTAÇÃO E SISTEMAS DE INFORMAÇÃO**

**Linguagem de Programação I**

**Aula 06: BUSCA EM LISTAS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | http://images.flatworldknowledge.com/ketchen/ketchen-fig05_x001.jpg | Nossos objetivos nesta aula são:   * Conhecer o problema da busca em listas. * Implementar as estratégias de busca sequencial e busca binária iterativas. | | Macintosh HD:Users:anacris:Desktop:Captura de Tela 2017-02-19 às 18.57.35.png | A referência para esta aula está no **Capítulo 4** do livro:  DIERBACH, C. *Introduction to Computer Science Using Python: A Computational Problem Solving Focus.* 1st Edition, New York: Wiley, 2012. | |

# RECURSOS PARA BUSCAR UM ELEMENTO EM UMA lista EM PYTHON

Na aula sobre LISTAS EM PYTHON já estudamos alguns recursos em Python que permitem realizar a busca de um elemento em uma lista.

Para verificar se um elemento está ou não em uma lista (ou em outro tipo de sequência), podemos utilizar a operação de *membership*, utilizando a sintaxe:

elemento in s

Exemplos:

string = 'Eu estou bem'  
tupla = (40, 33, 22, 53, 98, 10)  
lista = [25, 44, 99]  
print("letra 'e' está na string:", 'e' in string)  
print('número 22 está na tupla:', 22 in tupla)  
print('número 100 está na lista:', 100 in lista)

Para saber o índice onde há a primeira ocorrência do elemento na lista (ou em outro tipo de sequência), podemos utilizar o método *index*, utilizando a sintaxe:

s.index(elemento)

Exemplos:

string = 'Eu estou bem'  
tupla = (40, 33, 22, 53, 98, 10)  
lista = [25, 44, 99]  
print("índice da primeira ocorrência da letra 'e' na string:", string.index('e'))  
print('índice da primeira ocorrência do número 22 na tupla:', tupla.index(22))  
print('índice da primeira ocorrência do número 44 na lista:', lista.index(44))

Se procurarmos por um elemento que não está na lista, isto irá gerar um erro (**ValueError**) que interromperá a execução.

# IMPLEMENTAÇÃO DE ALGORITMOS de BUSCA

Apesar da linguagem Python já possuir as operações para a busca de elementos em uma lista, é necessário que um estudante da área de Computação entenda o funcionamento dos algoritmos de busca. Por este motivo, nesta aula você irá implementar os algoritmos de busca e não utilizará os recursos (*membership* e *index*) já existentes em Python para efetuar esta operação.

# PROBLEMA DA BUSCA EM LISTAS

Dada uma lista L e um elemento E, o problema da busca consiste em verificar se E pertence à lista L e, normalmente, quer se saber também qual é primeira posição em que o elemento E aparece na lista. Para isto, vamos definir uma operação chamada **search**(L,E), que devolve a posição da primeira ocorrência de E (caso exista na lista L) ou -1, caso E não pertença à lista.

No exemplo da lista L abaixo, **search**(L,7)=1 e **search**(L,53)=-1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 8 | 7 | 5 | 6 | 7 | -3 | 4 |

# BUSCA SEQUENCIAL OU LINEAR

A estratégia mais elementar para se implementar a operação **search** é a chamada **busca sequencial ou linear**. Na busca sequencial, parte-se da primeira posição da lista e, a cada elemento consultado, verifica-se se ele é o elemento procurado: caso afirmativo, devolve-se o índice; caso negativo, isto é, o elemento não está na lista, devolve-se -1.

A sequência abaixo ilustra a busca pelo elemento 6 em uma lista L:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 8 | 7 | 5 | 6 | 7 | -3 | 4 |
| **?** |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 8 | 7 | 5 | 6 | 7 | -3 | 4 |
|  | **?** |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 8 | 7 | 5 | 6 | 7 | -3 | 4 |
|  |  | **?** |  |  |  |  |
|  |  |  |  |  |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 8 | 7 | 5 | 6 | 7 | -3 | 4 |
|  |  |  | **?** |  |  |  |

Pela simulação da busca sequencial acima, concluímos que **search(L,6)=3**.

O algoritmo vinculado à operação de busca sequencial iterativo é mostrado abaixo:

**Algoritmo** linearSearch(L,e):

1. n = len(L)

**2. para** i=0 até n-1 **faça**

3. **se** L[i]=e

4. **então** **retorne** i

5. **retorne** -1

No pior caso, o elemento procurado não se encontra na lista. Assim, neste caso, faremos **n** comparações até chegar a esta posição. Dessa forma, dizemos que a busca sequencial tem complexidade ou desempenho **O(n)**.

# EXERCÍCIO tutoriado

A partir do esqueleto de código abaixo, implemente em Python a função de busca sequencial ou linear **linearSearch**. Invoque a sua função no programa principal:

import random  
# Gera uma lista com n elementos aleatórios de 1 a 100  
def generateList(n):  
 lista = []  
 for i in range(n):  
 lista.append(random.randint(1,100))  
 return lista  
   
# Verifica se o elemento está na lista L utilizando busca linear iterativa  
def linearSearch(L, e):

# Programa principal

QUESTÃO: é possível melhorar o algoritmo de busca sequencial se supormos que a lista esteja ordenada e que não contenha elementos repetidos?

# BUSCA BINÁRIA

Quando a lista está **ordenada** e não contém elementos repetidos, podemos utilizar uma técnica de divisão-e-conquista chamada **busca binária**:

* dividimos a lista ao meio
* se o elemento procurado for o elemento do meio, pare; senão, se o elemento procurado for menor que o elemento do meio, repetimos o processo para os elementos menores que o elemento do meio; caso contrário, repetimos o processo nos elementos maiores que o elemento do meio.

A simulação abaixo ilustra a busca binária para o elemento 7:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **low** |  |  |  |  |  | **high** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| -3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |
| **low** |  |  | **mid** |  |  | **high** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| -3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |
|  |  |  |  | **low** |  | **high** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| -3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |
|  |  |  |  | **low** | **mid** | **high** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| -3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |
|  |  |  |  | **low=high** |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| -3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |
|  |  |  |  | **low=high** |  |  |
|  |  |  |  | **mid** |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| -3 | 4 | 5 | 6 | 7 | 8 | 9 |

O algoritmo vinculado à operação de busca binária é mostrado abaixo:

Algoritmo binarySearch(L,e):

1. low=0

2. high=len(L)-1

3. enquanto (high>=low) faça

4. mid = (low+high)/2

5. se L[mid]==e

6. retorne mid

7. senão se L[mid] < e

8. low=mid+1

9. senão high=mid-1

10. fimenquanto

11. retorne -1

Algoritmo binarySearchR(L,e,low,high):

1. se low > high // elemento não encontrado

2. então retorne -1

3. mid=(low+high)/2

4. se L[mid]=e

5. **retorne** mid

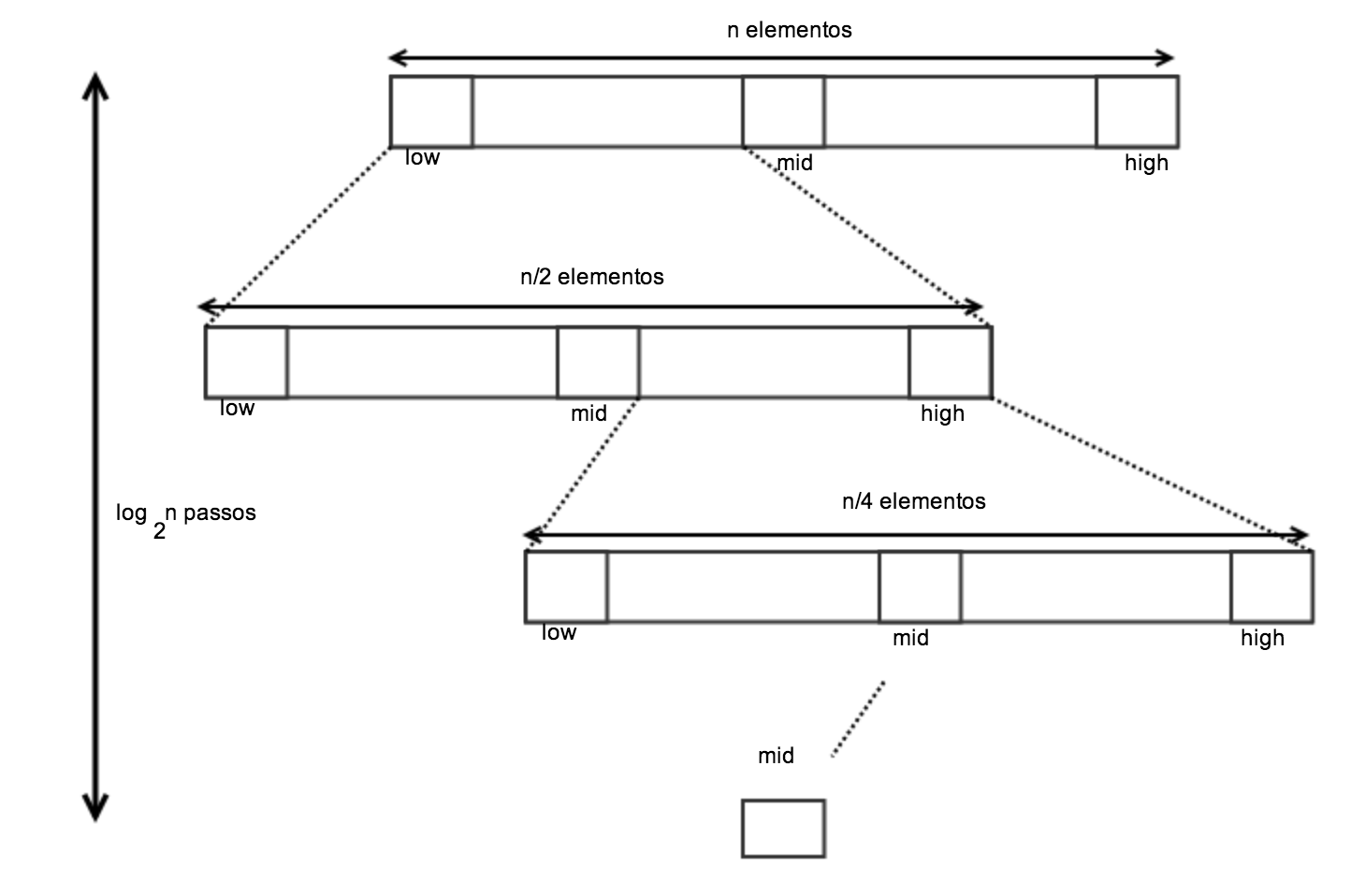
6. senão se e < L[mid]

7. **retorne** binarySearchR(L,e,low,mid-1) //Primeira metade

8. senão

9.  **retorne** binarySearchR(L,e,mid+1,high) //Segunda metade

Esquematicamente, a ação do algoritmo de busca binária é mostrada abaixo:



A cada passo do algoritmo, o tamanho da lista é reduzida aproximadamente pela metade. No pior caso, quando o elemento não está na lista, a redução da lista vai até o tamanho 1.

A partir deste ponto, low > high e o algoritmo termina.

Para se atingir o tamanho 1, devemos calcular a quantidade de passos k para atingir tal tamanho:

Como em cada passo as operações de comparação e divisão pela metade podem ser feitas em tempo **O(1)**, a complexidade total da busca binária é . Normalmente, omite-se a base 2 e é comum dizermos que este algoritmo é .

Como é menor que , dizemos que a busca binária, quando o vetor está ordenado e não contém elementos repetidos, é mais eficiente que a busca sequencial. Está análise será retomada e detalhada na disciplina de Estrutura de Dados.

# EXERCÍCIO COM DISCUSSÃO

A partir do esqueleto de código abaixo, implemente em Python a função de busca binária:

import random  
# Gera uma lista com n elementos aleatórios de 1 a 100, com valores

# distintos e ordenados  
def generateOrderedList(n):

# obtém n amostras distintas entre 1 e 100  
 lista = random.sample(range(1,101), n)

# neste exercício estamos utilizando o método sort para a ordenação,

# mas quando estudarmos ordenação não iremos utilizá-lo  
 lista.sort()

return lista

# Verifica se o elemento está na lista L utilizando busca binária iterativa  
def binarySearch(L, e):

# Verifica se o elemento está na lista L utilizando busca binária iterativa  
def binarySearchR(L, e, low, high):

# Programa principal

# EXERCÍCIOS

1) Na busca linear implementada em aula, o algoritmo tentava encontrar a primeira ocorrência do elemento procurado. Queremos agora encontrar a última ocorrência (caso exista) de um determinado elemento. Para isto, desenvolva a função:

def linearSearchLast(L, e):

que devolva, caso exista, o último índice ocupado pelo elemento **e** na lista **L**. Caso o elemento não exista na lista, a função deverá retornar -1.

2) Modifique o programa de busca sequencial para contar quantas comparações são feitas durante a busca. Teste a sua modificação e conte quantas operações são feitas para listas de tamanho 10, 50, 100, 500, 1000, 5000, 10000, 50000 e 100000. Gere um gráfico **tamanho da lista x número de comparações**.

3) Modifique o programa de busca binária para contar quantas comparações são feitas durante a busca. Teste a sua modificação e conte quantas operações são feitas para listas de tamanho 10, 50, 100, 500, 1000, 5000, 10000, 50000 e 100000 . Gere um gráfico **tamanho da lista x número de comparações**.