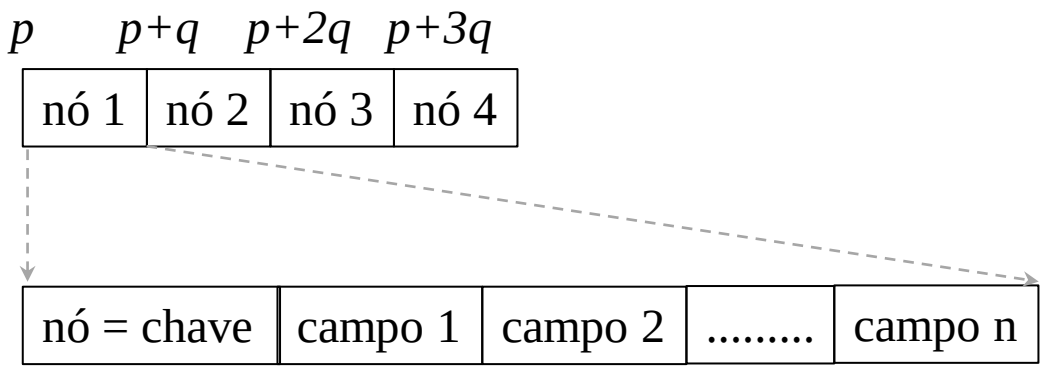
Complexidade dos algoritmos de busca

profº Mauricio Conceição Mario

Listas lineares em alocação sequencial

→ Uma ***lista linear*** é caracterizada pelo fato de seus nós estarem em posições contíguas, ou seja, o endereço do (*j* + *1*)-*ésimo* nó da lista se encontra ***q*** unidades adiante do nó correspondente ao *j-ésimo*. A constante ***q*** é o número de palavras de memória que cada nó ocupa. Exemplo:

p = posição de memória



(Szwarcfiter & Markenzon, 2020)

Busca em listas lineares em alocação sequencial → manipulação:

→ Seja uma ***lista linear L*** com *n* elementos. Modificação do algoritmo anterior de modo que o elemento procurado, pertencente ou não aos nós da lista, sempre seja adicionado ao final da mesma, evitando o segundo teste e diminuindo o tempo de execução:

```
L = [chave 1, campo 1, campo 2, ...,campo n, chave 2, ...]
n = dimensão (L)
        função busca_chave(nó)
            i := 1
            L = [chave 1, campo 1, campo 2, ...,campo n, chave 2, ..., nó]
            enquanto (L[i] != nó ) fazer:
                i := i + 1
            se (i != n+ 1) fazer:
                    retorna i
            senão
                    retorna 0
busca_chave(x)
```

A função *busca_chave* retorna o índice do nó inserido como argumento; o algoritmo executa somente os testes: enquanto($L[i] \neq nó$) e se ($i \neq n + 1$): não executa o senão pois o nó sempre será adicionado à lista, e portanto será encontrado.

A complexidade de pior caso para o algoritmo pode ser representada pela notação ***O*** (utilizada para limite assintótico superior), e para este algoritmo sua ordem seria → $O(n)$.

(Szwarcfiter & Markenzon, 2020)

C:\Users\cmari\.spyder-py3\Lista_linear_sequencial_II.py

.py | grafico_4.py | history_internal.py | função_enumerate.py | iteradores.py | inverte_sequência.py | datetime.py | soma_matricial.py | produto_matricial_3x3.py | Lista_linear_sequencial_II.py

Lista linear em alocação sequencial → algoritmo de busca otimizado, executado 1 vez

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jul 13 19:42:53 2022

@author: cmari
"""
import time

L = [1, "Goiânia", "Goiás", 2, "Bonito", "Mato Grosso", 3, "Carangola", "Minas Gerais", 4, "Peruíbe", "São Paulo", 20, "Serra", "Espírito Santo"]
n = len(L)
print ("dimensão da lista L = ", n)

print("\n")
def busca_linear(x):
  for p in range(1):
        L.append(x)
        i = 0
        #print ("dimensão da lista L = ", n)
        while L[i] != x:
            i = i + 1
            #print("i =", i)
        if i != n + 1:
            return i
        else:
            return "sem a chave"
tempo_inicial = time.time()
print("índice da chave (5) = ", busca_linear(5))
print("índice da chave (1) = ", busca_linear(1))
print("índice da chave (4) = ", busca_linear(4))
print("índice da chave (3) = ", busca_linear(3))
print("índice da chave (2) = ", busca_linear(2))
print("índice da chave (0) = ", busca_linear(0))
print("índice da chave (20) = ", busca_linear(20))
tempo_final = time.time()
print("tempo de execução = ", tempo_final - tempo_inicial)
```

Console 1/A

```
In [6]: runfile('C:/Users/cmari/.spyder-py3/Lista_linear_sequencial_II.py',
wdir='C:/Users/cmari/.spyder-py3')
dimensão da lista L =  15


índice da chave (5) =  15
índice da chave (1) =  0
índice da chave (4) =  9
índice da chave (3) =  6
índice da chave (2) =  3
índice da chave (0) =  20
índice da chave (20) =  12
tempo de execução =  0.0
```

n inicial = 15 + 5

n = 15
x = 5
L = [...., 5] ⇐ append(5): L[15]= 5 e n =15
enquanto (L[i] ≠ x) ⇒ L[0] = 1
  i = i + 1 ⇒ i = 1..., 14
  i = 15 ⇒ L[i] = 5
se (i ≠ n + 1) ⇒ (i = 15 e n + 1 = 16)
  retorna i = 15
senão
  retorna 0

n = 16
x = 1
L = [1,..., 5, 1] ⇐ append(1): L[16]= 1 e n =16

enquanto (L[i] ≠ x) ⇒ L[0] = 1
  i = 0
se (i ≠ n + 1) ⇒ se (i ≠ 17)
  retorna i = 0
senão
  retorna 0

Lista linear em alocação sequencial → algoritmo de busca otimizado, executado n vezes

C:\Users\cmari\.spyder-py3\Lista_linear_sequencial_II.py

Lista_linear_sequencial_II.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jul 13 19:42:53 2022

@author: cmari
"""
import time

L = [1, "Goiânia", "Goiás", 2, "Bonito", "Mato Grosso", 3, "Carangola", "Minas Gerais", 4, "Peruíbe", "São Paulo", 2
n = len(L)
print ("dimensão da lista L = ", n)

print("\n")
def busca_linear(x):
  for p in range(1000):
        L.append(x)
        i = 0
        #print ("dimensão da lista L = ", n)
        while L[i] != x:
            i = i + 1
            #print("i =", i)
        if i != n + 1:
            return i
        else:
            return "sem a chave"
tempo_inicial = time.time()
print("índice da chave (5) = ", busca_linear(5))
print("índice da chave (1) = ", busca_linear(1))
print("índice da chave (4) = ", busca_linear(4))
print("índice da chave (3) = ", busca_linear(3))
print("índice da chave (2) = ", busca_linear(2))
print("índice da chave (0) = ", busca_linear(0))
print("índice da chave (20) = ", busca_linear(20))
tempo_final = time.time()
print("tempo de execução = ", tempo_final - tempo_inicial)

```

Console 1/A

```
In [7]: runfile('C:/Users/cmari/.spyder-py3/Lista_linear_sequencial_II.py',
wdir='C:/Users/cmari/.spyder-py3')
dimensão da lista L =  15


índice da chave (5) =  15
índice da chave (1) =  0
índice da chave (4) =  9
índice da chave (3) =  6
índice da chave (2) =  3
índice da chave (0) =  20
índice da chave (20) =  12
tempo de execução =  0.0010340213775634766
```

Comparação para os tempos de execução entre a versão do algarismo de busca linear que executa dois testes e a versão que insere incondicionalmente a chave ou nó procurado no fim da lista:

```
def busca_linear(x):
  for p in range(1000):
      i = 0
      while i < n:
        if L[i] == x:
          return i
          #i = n + 1
        else:
          i = i + 1
tempo_inicial = time.time()
print("índice da chave (5) = ", busca_linear(5))
print("índice da chave (1) = ", busca_linear(1))
print("índice da chave (4) = ", busca_linear(4))
print("índice da chave (3) = ", busca_linear(3))
print("índice da chave (2) = ", busca_linear(2))
print("índice da chave (0) = ", busca_linear(0))
print("índice da chave (20) = ", busca_linear(20))
tempo_final = time.time()
print("tempo de execução = ", tempo_final - tempo_inicial)
```

```
dimensão da lista L =  15
L[0] 1
L[3] 2
L[6] 3
L[9] 4
L[4] Bonito
L[4][2] n
L[2][1] o


índice da chave (5) =  None
índice da chave (1) =  0
índice da chave (4) =  9
índice da chave (3) =  6
índice da chave (2) =  3
índice da chave (0) =  None
índice da chave (20) =  12
tempo de execução =  0.006198406219482422
```

```
tempo de execução =  0.006198406219482422
```

← Algoritmo executa dois testes

```
tempo de execução =  0.0010340213775634766
```

← Algoritmo otimizado com adição de nó procurado

Análise da complexidade dos algoritmos de busca em listas de alocação sequencial

→ A ***complexidade de pior caso*** para os algoritmos de busca pode ser representada pela notação ***O*** (utilizada para limite assintótico superior), com ordem → $\boldsymbol{O}(n)$.

→ `tempo de execução =  0.0061984062194824 22` ← Algoritmo executa dois testes

→ `tempo de execução =  0.0010340213775634766` ← Algoritmo otimizado com adição de nó procurado

→ Seja q a probabilidade de sucesso no resultado da busca. Considerando a ***complexidade média***, é relevante considerar que entradas distintas que tenham a chave procurada na mesma posição podem ser consideradas como idênticas.

(Szwarcfiter & Markenzon, 2020)

Análise da complexidade média dos algoritmos de busca em listas de alocação sequencial

A análise de *complexidade de pior caso* é equivalente para estes algoritmos de busca; a análise da *complexidade média* é mais eficiente neste caso.

→ Seja q a probabilidade de sucesso no resultado da busca. Considerando a ***complexidade média***, é relevante considerar que entradas distintas que tenham a chave procurada na mesma posição podem ser consideradas como idênticas.

→ A ***complexidade média*** é dada por: $\Sigma \; p(E_k) \; t(E_k)$, ou seja, a somatória das probabilidades das entradas $p(E_k)$ e o número de passos para esta entrada, $t(E_k)$.

→ Seja $E_i \quad 1 \leq i \leq n$ uma entrada em que a chave procurada ocupa a *i-ésima* posição da lista, e E_0 a entrada que corresponde à busca sem sucesso. Então, as probabilidades das entradas são: $p(E_k) = q/n, \qquad 1 \leq k \leq n$

$$p(E_0) = 1 - q$$

(Szwarcfiter & Markenzon, 2020)

Análise da complexidade média dos algoritmos de busca em listas de alocação sequencial

→ O número total de passos efetuados pelo algoritmo será:

$$t(E_k) = k, \qquad 1 \leq k \leq n$$
$$t(E_0) = n$$

→ A expressão da ***complexidade média*** será:

$$\Sigma_{0 \leq k \leq n}\ p(E_k)\ t(E_k) = (1- q)n + \Sigma_{1 \leq k \leq n}\ (q/n)k$$

Para a primeira versão do algoritmo de busca, implementações com essas variações são demonstradas a seguir:

(Szwarcfiter & Markenzon, 2020)

Lista linear em alocação sequencial → algoritmo de busca com a chave na primeira posição

iteradores.py | inverte_sequência.py | datetime.py | soma_matricial.py | produto_matricial_3x3.py | Lista_linear_sequencial_II.py | Lista_linear_sequencial_I.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jul 13 19:42:53 2022

@author: cmari
"""
import time

L = [1, "Goiânia", "Goiás", 2, "Bonito", "Mato Grosso", 3, "Carangola", "Minas Gerais", 4, "Peruíbe", "São Paulo", 20, "Se
n = len(L)
print ("dimensão da lista L = ", n)
#print ("L[0]", L[0])
#print ("L[3]", L[3])
#print ("L[6]", L[6])
#print ("L[9]", L[9])
#print ("L[4]", L[4])
#print ("L[4][2]", L[4][2])
#print ("L[2][1]", L[2][1])
print("\n")
def busca_linear(x):
  for p in range(1000):
       i = 0
       while i < n:
         if L[i] == x:
           return i
           #i = n + 1
         else:
            i = i + 1
tempo_inicial = time.time()
#print("índice da chave (5) = ", busca_linear(5))
print("índice da chave (1) = ", busca_linear(1))
#print("índice da chave (4) = ", busca_linear(4))
#print("índice da chave (3) = ", busca_linear(3))
#print("índice da chave (2) = ", busca_linear(2))
#print("índice da chave (0) = ", busca_linear(0))
#print("índice da chave (20) = ", busca_linear(20))
tempo_final = time.time()
print("tempo de execução = ", tempo_final - tempo_inicial)
```

Console 1/A

```
Python 3.8.3 (default, Jul  2 2020, 17:30:36) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.16.1 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/cmari/.spyder-py3/Lista_linear_sequencial_I.py',
wdir='C:/Users/cmari/.spyder-py3')
dimensão da lista L =  15


índice da chave (1) =  0
tempo de execução =  0.0
```

Lista linear em alocação sequencial → algoritmo de busca com a chave na 12ª posição

iteradores.py | inverte_sequência.py | datetime.py | soma_matricial.py | produto_matricial_3x3.py | Lista_linear_sequencial_II.py | Lista_linear_sequencial_I | Lista_linear_sequencial_I.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jul 13 19:42:53 2022

@author: cmari
"""
import time

L = [1, "Goiânia", "Goiás", 2, "Bonito", "Mato Grosso", 3, "Carangola", "Minas Gerais", 4, "Peruíbe", "São Paulo", 20, "Se
n = len(L)
print ("dimensão da lista L = ", n)
#print ("L[0]", L[0])
#print ("L[3]", L[3])
#print ("L[6]", L[6])
#print ("L[9]", L[9])
#print ("L[4]", L[4])
#print ("L[4][2]", L[4][2])
#print ("L[2][1]", L[2][1])
print("\n")
def busca_linear(x):
  for p in range(1000):
        i = 0
        while i < n:
          if L[i] == x:
            return i
            #i = n + 1
          else:
             i = i + 1
tempo_inicial = time.time()
#print("índice da chave (5) = ", busca_linear(5))
#print("índice da chave (1) = ", busca_linear(1))
#print("índice da chave (4) = ", busca_linear(4))
#print("índice da chave (3) = ", busca_linear(3))
#print("índice da chave (2) = ", busca_linear(2))
#print("índice da chave (0) = ", busca_linear(0))
print("índice da chave (20) = ", busca_linear(20))
tempo_final = time.time()
print("tempo de execução = ", tempo_final - tempo_inicial)
```

Console 1/A

```
In [2]: runfile('C:/Users/cmari/.spyder-py3/Lista_linear_sequencial_I.py',
wdir='C:/Users/cmari/.spyder-py3')
dimensão da lista L =  15


índice da chave (20) =  12
tempo de execução =  0.0
```

Lista linear em alocação sequencial → algoritmo de busca com a chave não encontrada


```python
# -*- coding: utf-8 -*-
"""
Created on Wed Jul 13 19:42:53 2022

@author: cmari
"""
import time

L = [1, "Goiânia", "Goiás", 2, "Bonito", "Mato Grosso", 3, "Carangola", "Minc
n = len(L)
print ("dimensão da lista L = ", n)
#print ("L[0]", L[0])
#print ("L[3]", L[3])
#print ("L[6]", L[6])
#print ("L[9]", L[9])
#print ("L[4]", L[4])
#print ("L[4][2]", L[4][2])
#print ("L[2][1]", L[2][1])
print("\n")
def busca_linear(x):
  for p in range(1000):
       i = 0
       while i < n:
         if L[i] == x:
           return i
           #i = n + 1
         else:
            i = i + 1
tempo_inicial = time.time()
print("índice da chave (5) = ", busca_linear(5))
#print("índice da chave (1) = ", busca_linear(1))
#print("índice da chave (4) = ", busca_linear(4))
#print("índice da chave (3) = ", busca_linear(3))
#print("índice da chave (2) = ", busca_linear(2))
#print("índice da chave (0) = ", busca_linear(0))
#print("índice da chave (20) = ", busca_linear(20))
tempo_final = time.time()
print("tempo de execução = ", tempo_final - tempo_inicial)
```

Console 1/A

```
In [3]: runfile('C:/Users/cmari/.spyder-py3/Lista_linear_sequencial_I.py',
wdir='C:/Users/cmari/.spyder-py3')
dimensão da lista L =  15


índice da chave (5) =  None
tempo de execução =  0.0016710758209228516
```

***Exercício*:**

Comparar a pesquisa nas mesmas posições com o algoritmo otimizado que inclui o nó pesquisado ao final da lista.

Referências Bibliográficas

- Estruturas de Dados e Seus Algoritmos
  Jayme L. Szwarcfiter & Lilian Markenzon
  3ª edição – editora *gen* LTC – 2010 - 2020

- Matemática Avançada para Engenharia
  Dennis G. Zill & Michael R. Cullen
  Álgebra Linear e Cálculo Vetorial (2) – 3ª edição, editora Bookman – 2009

- Algoritmos – Teoria e Prática
  Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
  3ª edição – editora Elsevier - *gen* LTC – 2012