Algoritmos e suas Complexidades

Diferentes Estratégias

Criando Algoritmos Eficientes

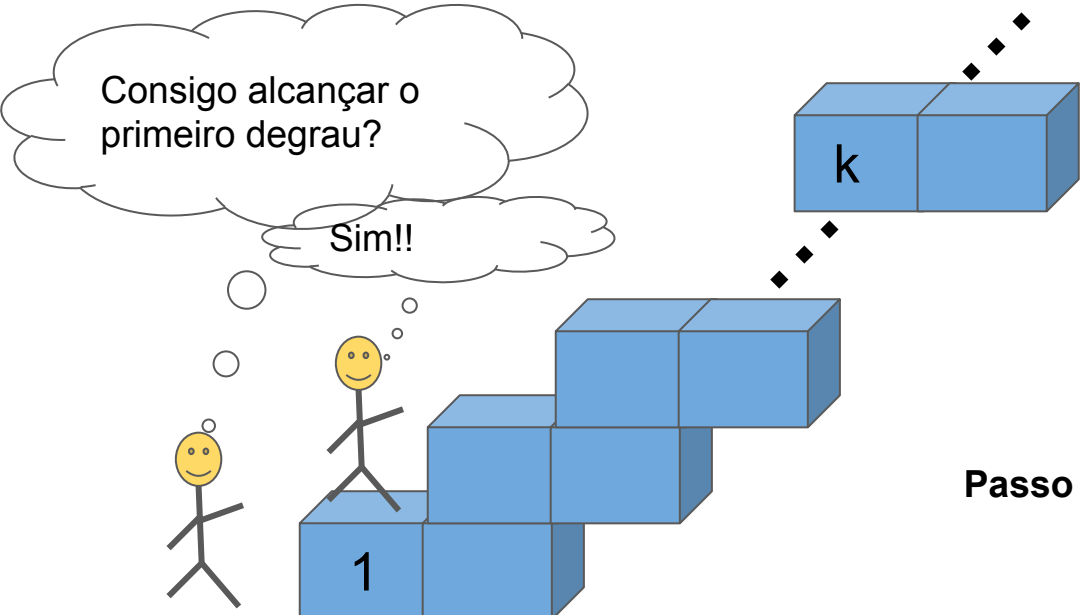
- Tarefa difícil de executar
  - Cientistas da computação podem criar, talvez, um algoritmo durante toda a sua carreira.
- Pensamos que criamos mas na verdade apenas reduzimos a complexidade do problema envolvido para problemas análogos já solucionados fazendo:
  - Uma análise da complexidade do problema a ser solucionado
  - Uma divisão do problema em subproblemas.
  - Relacionando tais subproblemas a problemas com algoritmos eficientes já criados.

Criando Algoritmos Eficientes

- Outro lado da moeda:
  - Nem sempre o algoritmo mais eficiente é escolhido
  - Tal algoritmo pode ser desnecessariamente difícil de entender ou desenvolver num primeiro momento.
- Uma boa estratégia:
  - Resolver o problema da forma mais direta possível.
  - Identificar possíveis gargalos computacionais.
  - Aprimorar a complexidade computacional para reduzir tais gargalos.

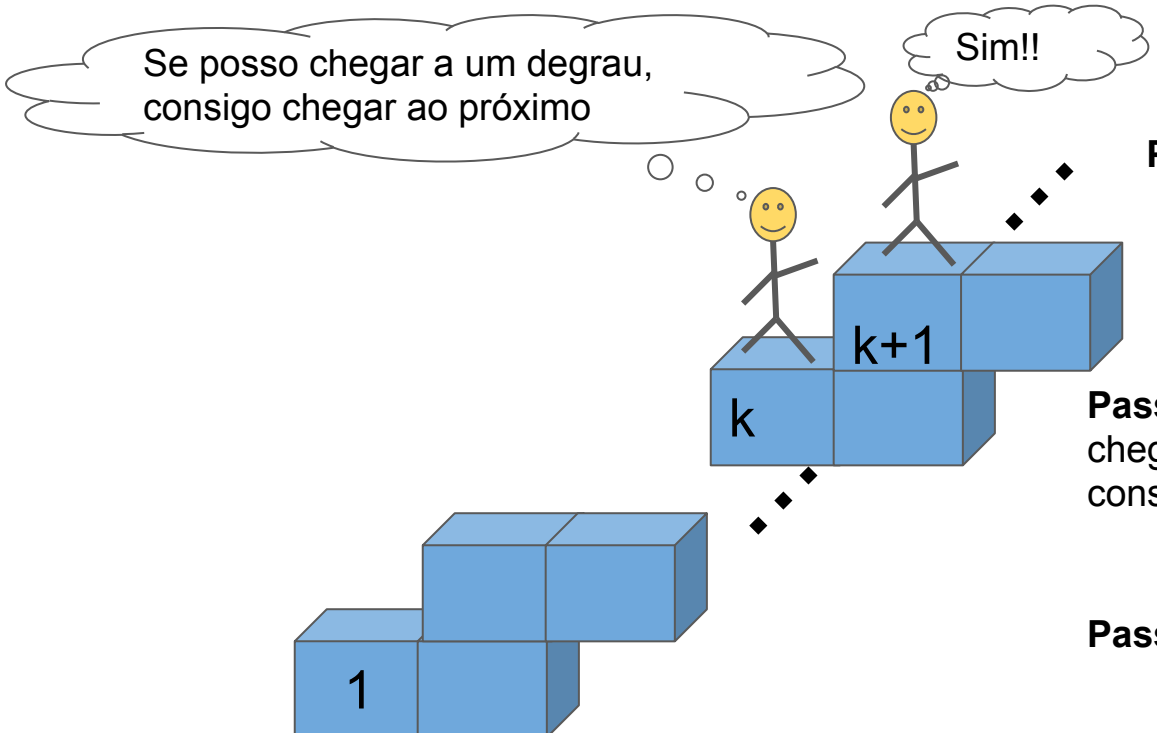
Conceitos Preliminares: Indução

Exemplo: Subindo uma escada infinita



Passo básico: Subir primeiro degrau

Conceitos Preliminares: Indução



Provado por indução !!!

Passo Indutivo: Se consegue chegar a um degrau qualquer, então consegue passar para o **próximo**

Passo básico: Subir primeiro degrau

Conceitos Preliminares: Indução

Primeiro Princípio da Indução:

Passo básico: P(1) é verdadeiro.

Passo indutivo: $\forall k \in Z_+$, $P(k) \longrightarrow P(k+1)$

$\Big\}$ P(n) verdade $\forall n \in Z_+$

Conceitos Preliminares: Indução

Exemplo 1: Prove que $1+3+5+...+(2n-1) = n^2$

Passo básico n=1:

$1 = 1^2 \Rightarrow 1 = 1$ Ok

Passo Indutivo:

Hipótese de Indução $n=k \Rightarrow 1+3+5+...+(2k-1) = k^2$ $n=k \Rightarrow 1+3+5+...+(2k-1) = k^2$

Precisamos provar para n = k+1, ou seja,

$1+3+5+...+(2k-1) + [2(k+1)-1] = (k+1)^2$ **????**

Conceitos Preliminares: Indução

$\mathbf{1+3+5+...+(2k-1)} + [2(k+1)-1] = (k+1)^2$

$\mathbf{k^2} + [2(k+1)-1] = (k+1)^2$, Hipótese de Indução: $1+3+5+...+(2k-1) = k^2$

$\mathbf{k^2} + [2(k+1)-1] = (k+1)^2$

$k^2 + 2k + 1 = (k+1)^2$

$(k+1)^2 = (k+1)^2$

Conceitos Preliminares: Indução

Segundo Princípio da Indução (Indução Forte):

Passo básico: P(1) é verdadeiro.

Passo indutivo: $\forall k \in Z_+ \quad [(P(r) \text{ com } 0 \leq r \leq k) \longrightarrow P(k+1)]$

$\Big\}$ P(n) verdade $\forall n \in Z_+$

Conceitos Preliminares: Indução

- A **corretude de um algoritmo** indica que ele termina sua execução, retornando saídas corretas para todas as instâncias do problema.
- Vamos provar a **corretude de algoritmos** usando **Indução Matemática**.

Conceitos Preliminares: Corretude de algoritmos

- **Invariante de laço**:
  - propriedade que é verdadeira cada vez que a condição do laço é avaliada.
  - Propriedade que é verdadeira antes e depois de cada iteração do laço.
- A propriedade de um invariante de laço é satisfeita independente de qual iteração do laço está sendo executada.

Conceitos Preliminares: Corretude de algoritmos

- **Invariante de laço** pode ser utilizada para determinar a **corretude de algoritmos**.
- Três aspectos precisam ser considerados:
  - **Inicialização**: Um invariante de laço é verdadeiro antes da primeira iteração do laço.
  - **Manutenção**: Se for verdadeiro antes de uma iteração do laço, ele permanece verdadeiro antes da próxima iteração.
  - T**erminação**: A invariante nos dá uma propriedade útil que ajuda a mostrar que o algoritmo está correto quando o laço termina.

Conceitos Preliminares: Corretude de algoritmos

- Estratégias para verificar tais aspectos:
  - **Estratégia ruim**: Verificar o comportamento após cada iteração.
  - **Estratégia boa**: Indução matemática.
    - **Passo base:** Provar que a hipótese de indução ocorre para os valores de entrada do laço.
    - **Passo Indutivo:** Provar que se a hipótese de indução ocorre após k iterações, ela também é verificada após k+1 iterações.
    - Utilize a hipótese de indução para comprovar que o algoritmo está correto ao final do laço.
- A **hipótese de indução** na **estratégia boa** é a **invariante de laço**!!!

Conceitos Preliminares: Corretude de algoritmos

Exemplo 1:

```
Algorithm Muito-Simples
        a←c;
        b←0;
        while (a>0)
                do   a←a - 1;
                     b←b + 1;
        return b;
```

- Qual a entrada?
- Qual a saída?
- Qual a propriedade do laço?

Conceitos Preliminares: Corretude de algoritmos

Exemplo 1:

```
Algorithm Muito-Simples
        a←c;
        b←0;
        while (a>0)
                do   a←a - 1;
                     b←b + 1;
        return b;
```

Verificando a corretude do algoritmo:

Invariante de laço: a+b = c

Passo base: Antes do início do laço, temos

a←c;

b←0;

Logo, a + b = c+ 0 = c ⇒ a+b = c Ok!!

Conceitos Preliminares: Corretude de algoritmos

Exemplo 1:

```
Algorithm Muito-Simples
    a←c;
    b←0;
    while (a>0)
        do   a←a - 1;
             b←b + 1;
    return b;
```

Verificando a corretude do algoritmo:

Invariante de laço: a+b = c

Passo Indutivo: Se a+b= c ocorre após k iterações, então a+b=c após k+1 iterações.

Prova:

Após a iteração k, por hipótese de indução, temos a=c-k e b=k com a+b=c;

Conceitos Preliminares: Corretude de algoritmos

Exemplo 1:

```
Algorithm Muito-Simples
    a←c;
    b←0;
    while (a>0)
        do   a←a - 1;
             b←b + 1;
    return b;
```

Verificando a corretude do algoritmo:

Passo Indutivo: Se a+b= c ocorre após k iterações, então a+b=c após k+1 iterações.

Prova:

Na iteração k+1, usando a H.I, temos:

$a \leftarrow a - 1 \Leftrightarrow a = c-k-1$

$b \leftarrow b + 1 \Leftrightarrow b = k+1$

Logo, $a + b = (c-k-1)+(k+1) = c$

$\Rightarrow a+b = c$ Ok!!

Conceitos Preliminares: Corretude de algoritmos

Exemplo 1:

```
Algorithm Muito-Simples
    a←c;
    b←0;
    while (a>0)
        do  a←a - 1;
            b←b + 1;
    return b;
```

Verificando a corretude do algoritmo:

Terminação: O algoritmo após o laço deve estar correto.
Prova: A condição do laço é violada quando a ← 0.
Pela propriedade da invariante de laço, temos b=c.
Logo,

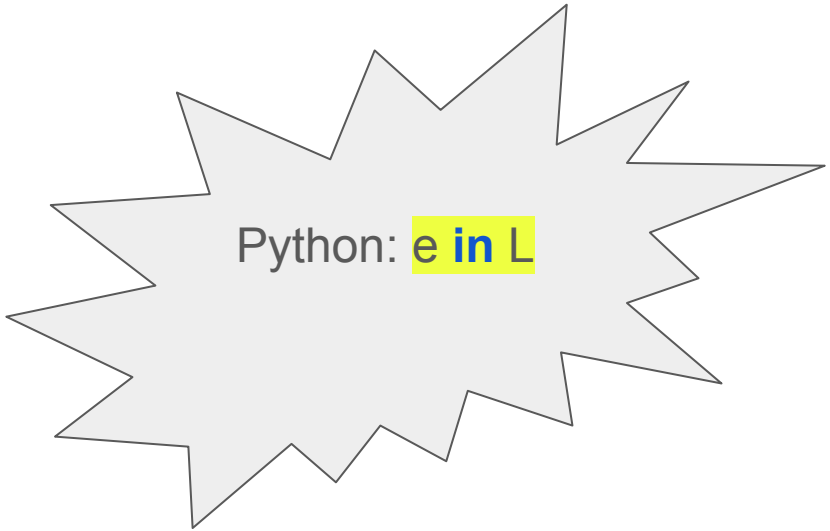
```
return b  //Retorna valor de c !!
```

Algoritmos de Busca

- Encontra um item ou grupo de itens dentro de um conjunto ou sequência de itens.
- Tal conjunto ou sequência de itens define o espaço de busca.
    - Conjunto de valores inteiros, base de dados com registros médicos, sequência de valores financeiros, etc.
    - Vários problemas reais podem ser reduzidos a um problema de busca.

Algoritmos de Busca

- Input:
  - (Lista **L** de itens, item **e** procurado)
- Saída:
  - **Verdadeiro**, se e está em **L**.
  - **Falso**; caso contrário



Algoritmos de Busca

```python
def busca(L,e):
    for i in range(len(L)):
        if L[i] == e:
            return True
    return False
```

- O(len(L)): complexidade linear considerando o tamanho de L, caso cada operação no laço seja executada em tempo constante.
- Python faz isso?

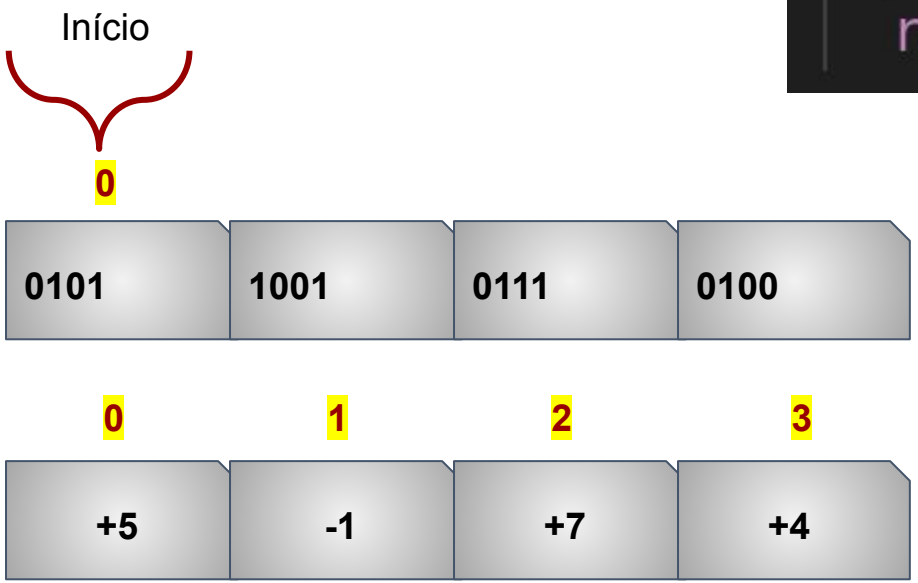
Algoritmos de Busca

```python
def busca(L,e):
    for i in range(len(L)):
        if L[i] == e:
            return True
    return False
```

- O(len(L)): complexidade linear considerando o tamanho de L, caso cada operação no laço seja executada em tempo constante.
- Python faz isso?
- Python acessa o i-ésimo elemento em tempo constante?

Algoritmos de Busca

```
def busca(L,e):
    for i in range(len(L)):
        if L[i] == e:
            return True
    return False
```



Localização: início +4i

i= 0, início = 0, temos 0 + 4(0) = 0
01011001011110100

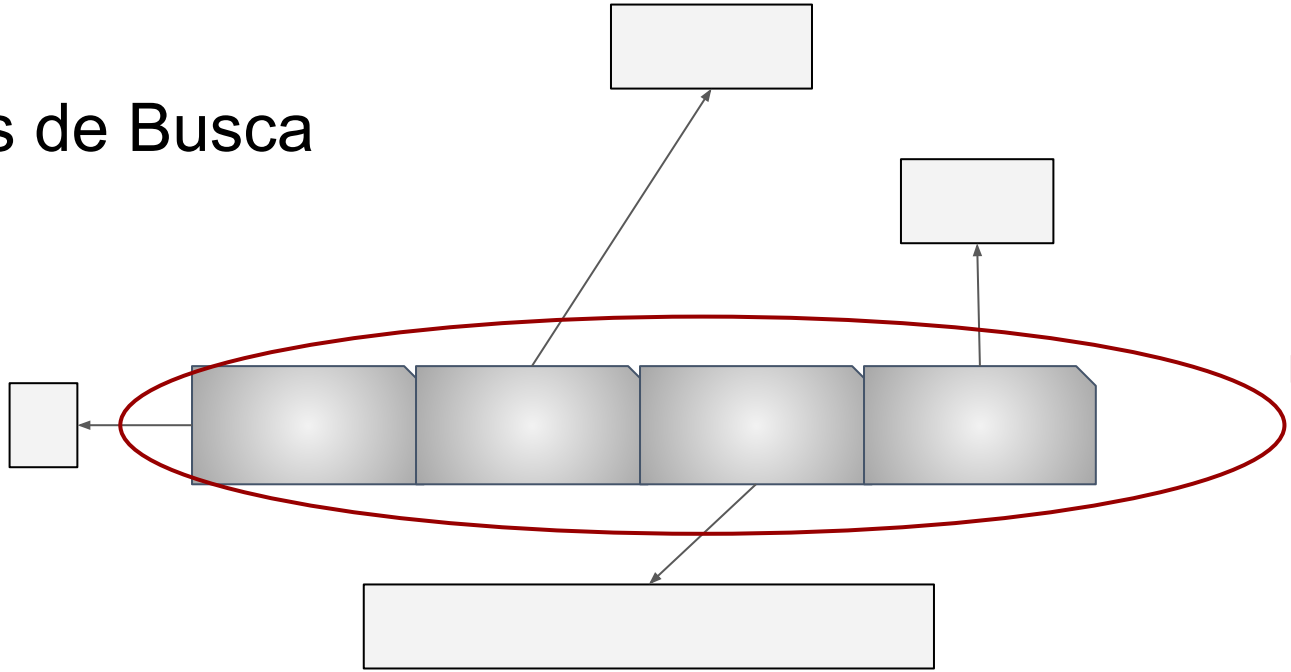
i= 1, start = 0, temos 0 + 4(1) = 4
0101**1**001011110100

i= 2, start = 0, temos 0 + 4(2) = 8
01011001**0**1110100

Algoritmos de Busca

- No exemplo anterior, cada elemento da lista tem o mesmo tamanho.
- O endereço em memória do elemento i da lista é simplesmente **início + 4i**.
- Python conseguiria calcular tal endereço em tempo constante.

Algoritmos de Busca



- A lógica anterior não muda, caso consideremos quatro unidades de memória ocupada por ponteiros na lista contendo mais quatro unidades de memória para armazenar endereços dos objetos referenciados.
- O endereço do i-ésimo elemento da lista será armazenado na posição **início + 4 + 4i**.
- Tal endereço continua sendo determinado em tempo constante.

Algoritmos de Busca

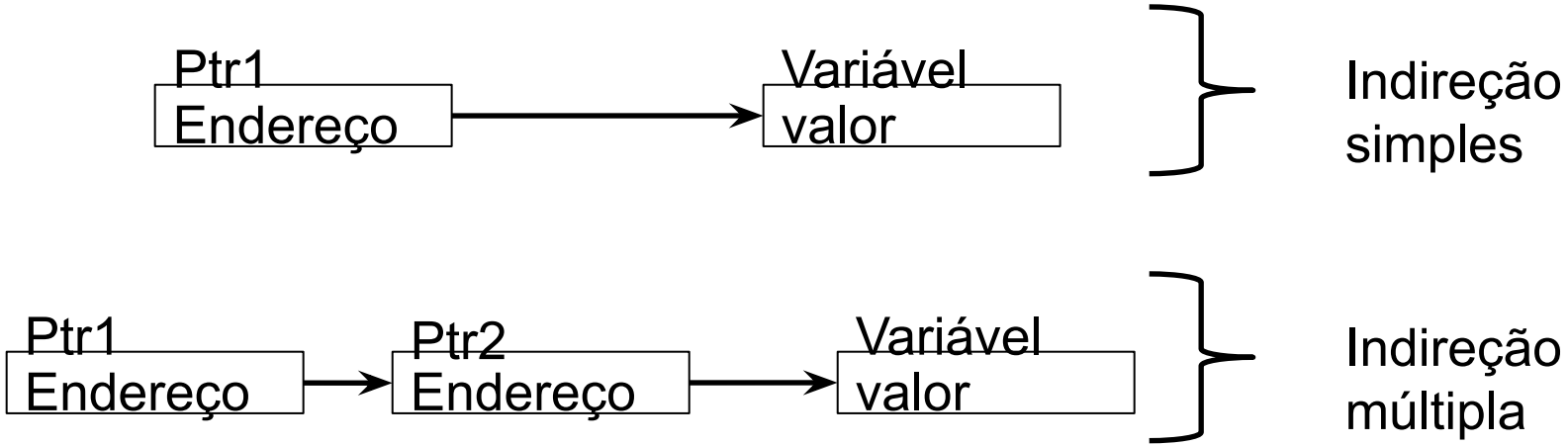
- O conceito de indireção foi utilizado, onde o acesso ocorre primeiro a uma referência do objeto final a ser obtido.
- Basicamente, utiliza-se uma variável para se referir ao objeto ao qual essa variável está vinculada.
- Temos a chamada indireção em dois níveis quando usamos uma variável para acessar uma lista e, em seguida, uma referência armazenada nessa lista para acessar outro objeto.

Algoritmos de Busca

- O conceito de indireção foi utilizado, onde o acesso ocorre primeiro a uma referência do objeto final a ser obtido.
- Basicamente, utiliza-se uma variável para se referir ao objeto ao qual essa variável está vinculada.
- Temos a chamada indireção em dois níveis quando usamos uma variável para acessar uma lista e, em seguida, uma referência armazenada nessa lista para acessar outro objeto.

Algoritmos de Busca

- Indireção múltipla ou ponteiros para ponteiros ocorre quando temos ponteiro apontando para outro ponteiro que aponta para um determinado valor.



Algoritmos de Busca

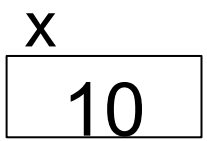
Exemplo em linguagem C

```
#include <stdio.h>
int main(void){
    int x, *p, **q;
    x=10;
    p=&x;
    q=&p;
    printf("%d",**q);
      return 0;
}
```

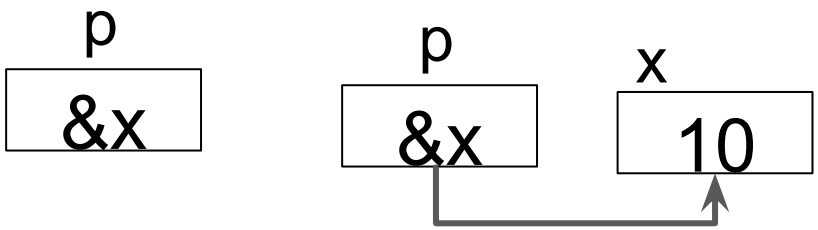
- Um ponteiro para um ponteiro deve ser declarado com a adição de mais um *.
  - int **q;
  - Indica que q é um ponteiro para um ponteiro do tipo int.
- **q
  - acessa o valor apontado pelos ponteiros.

Algoritmos de Busca

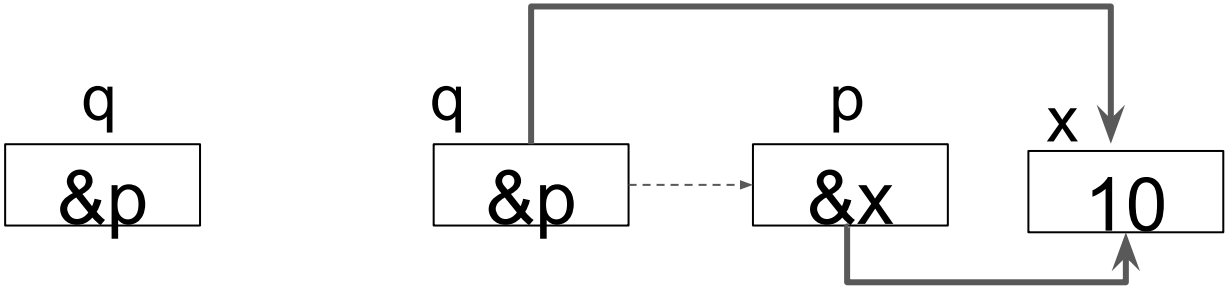
```
int x=10
```



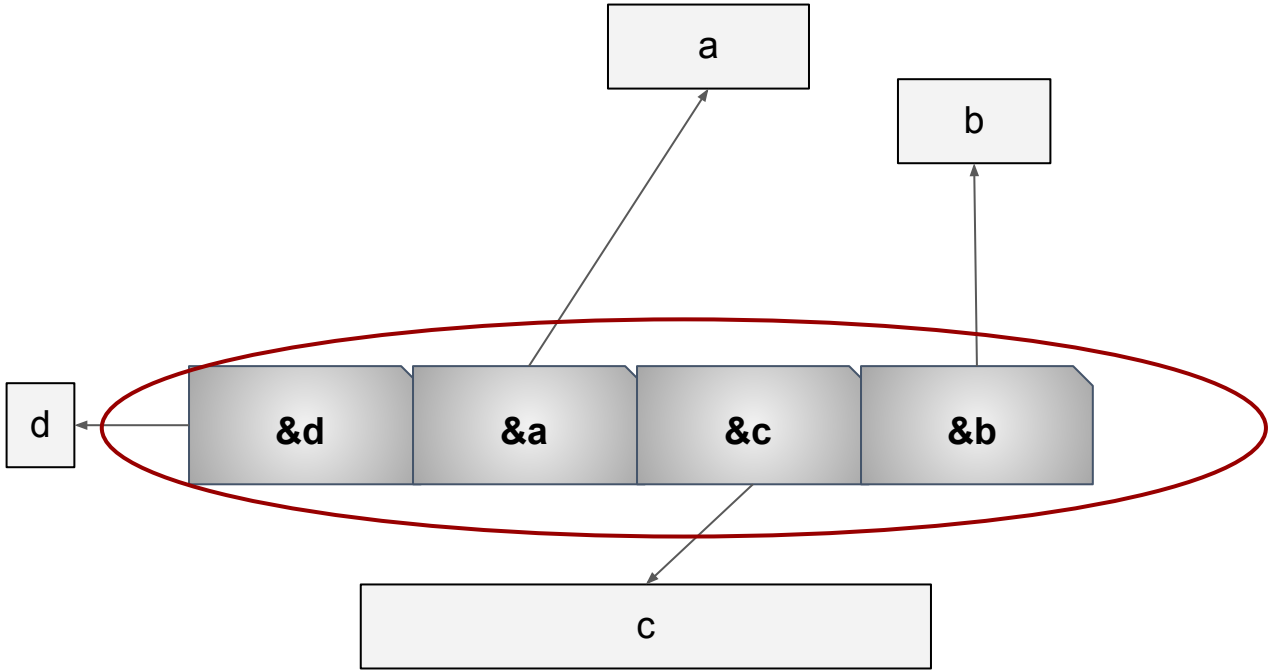

```
p=&x
```




```
q=&p
```



Algoritmos de Busca



Lista de ponteiros!!

Busca Binária

- O(len(L)) é o melhor que podemos alcançar em um algoritmo de busca?
  - Sim, se não há uma informação sobre como os elementos se relacionam e sobre a ordem do seu armazenamento na lista.
  - Precisamos verificar cada elemento de L no **pior caso!**

Busca Binária

Melhora o tempo médio mas continua O(len(L)) no pior caso.

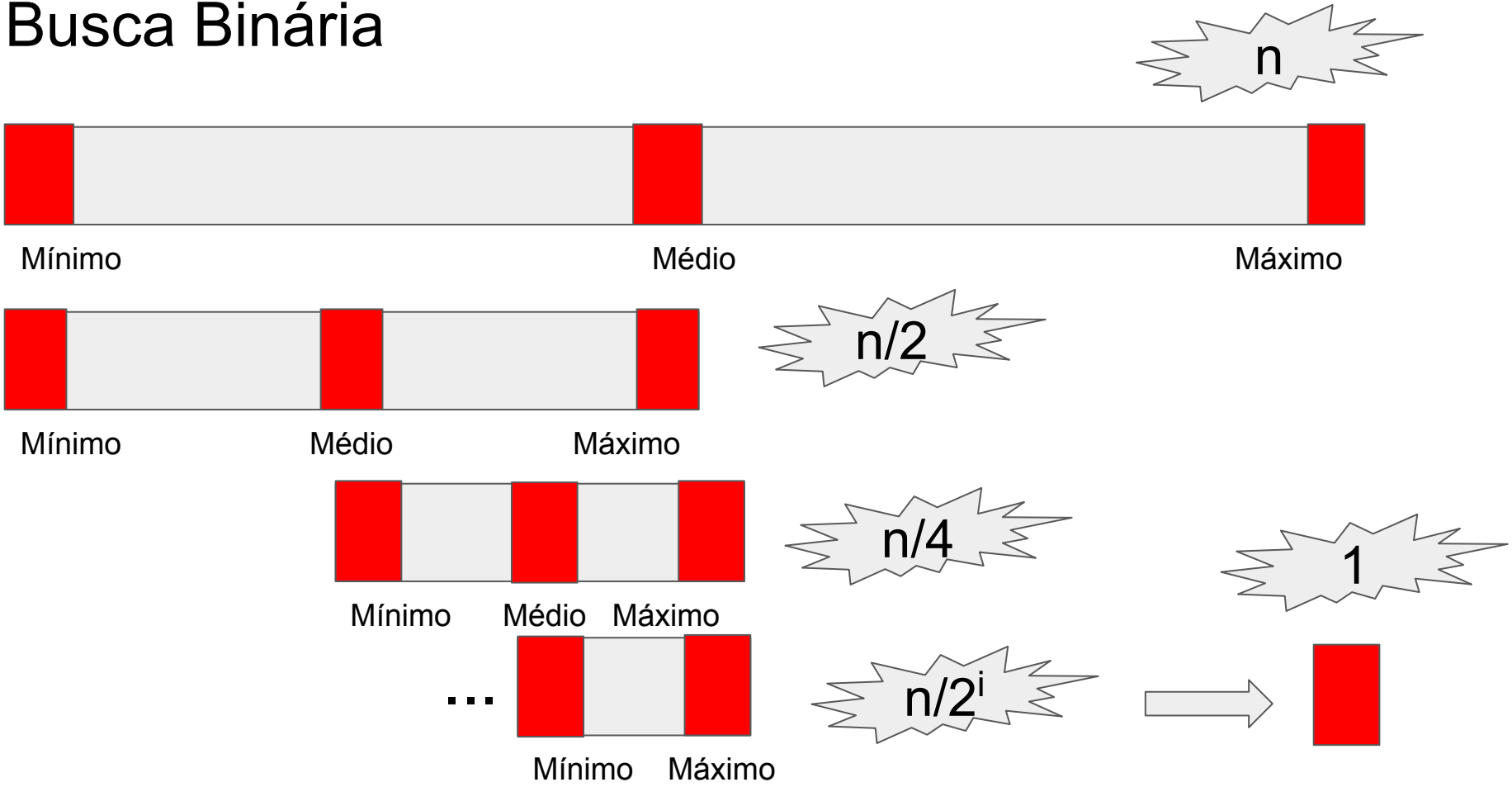
- Agora, vamos mudar o problema:
  - Input: (**L, e**) com elementos de **L** dispostos em ordem crescente.
  - Saída: Verdadeiro, se **e** está em **L**; Falso, caso contrário.

```
1   import random
2
3   def busca(L,e):
4       for i in range(len(L)):
5           if L[i] == e:
6               return True
7           if L[i] > e:
8               return False
9       return False
```

Busca Binária

- Outra estratégia:
  - Determine um índice para item mediano da lista, ou seja, dividir a lista aproximadamente pela metade.
  - Verifique se **L[i] == e**.
  - Caso contrário, verifique se L[i] é maior ou menor que **e**.
  - A partir da resposta, buscar **e** na metade inferior ou superior da lista.

Busca Binária



```python
def busca(L, e):
    """Entrada: lista L em ordem crescente.
       Saída: Verdadeiro, se e está em L; Falso, caso contrário."""

    def buscaBinaria(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid:
                #Nada para buscar
                return False
            else: return buscaBinaria(L, e, low, mid - 1)
        else:
            return buscaBinaria(L, e, mid + 1, high)

    if len(L) == 0:
        return False
    else:
        return buscaBinaria(L, e, 0, len(L) - 1)
```

Busca Binária

- A implementação assume que L está ordenada.
- A tarefa de verificar se essa condição foi satisfeita é do algoritmo principal **busca()**.
- Caso L não esteja ordenada, não há obrigação de que o resultado seja aquele esperado. Logo, podemos ter:
    - Elemento é localizado
    - Código “crasha”
    - Obtemos uma resposta errada.

Busca Binária

```
if len(L) == 0:
    return False
else:
    return buscaBinaria(L, e, 0, len(L) - 1)
```

- Devemos tratar isso no código?
  - Isso ajudaria a eliminar os problemas mencionados…
  - …ao custo O(len(L)) para verificar se a condição é válida, antes de executar a busca.
- **busca()** é um exemplo de função wrapper:
  - Fornece uma interface para o código do cliente.
  - Trata-se de uma passagem que não faz nenhuma computação séria.
  - Executa a função auxiliar **buscaBinaria()**.

Busca Binária

- Podemos eliminar **busca()** e usar apenas **buscaBinaria()**!!
  - **Não!!!** Parâmetros como **low** e **high** não tem nada haver com a abstração de procurar um elemento em uma lista.
  - Tais detalhes de implementação devem ser escondidos dos programas que chamam a busca binária.

Busca Binária

- A etapa de decrementar a função, ou seja, reduzir o parâmetro de entrada, tem as seguintes propriedades:
    1. Mapeia os valores para os quais os parâmetros ficam limitados a valores não negativos.
    2. Quando o valor é 0, o passo recursivo é encerrado.
    3. O valor de decremento é menor em cada chamada recursiva em relação à chamada anterior.

Busca Binária

- **buscaBinaria()** executa duas chamadas recursivas:
  - Uma chamada é ajustada com parâmetros para buscar na metade inferior da lista corrente.

```
17          else: return buscaBinaria(L, e, low, mid - 1)
```

  - Outra chamada é ajustada com parâmetros para buscar na metade superior da lista corrente.

```
18          else:
19              return buscaBinaria(L, e, mid + 1, high)
```

  - Em ambos os casos, a diferença **low-high** se reduz à metade.

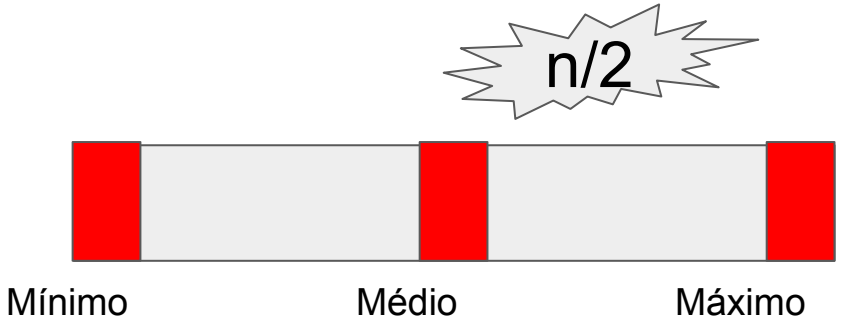
```
10          mid = (low + high)//2
```

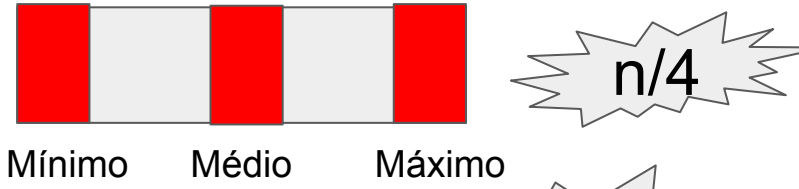
Busca Binária

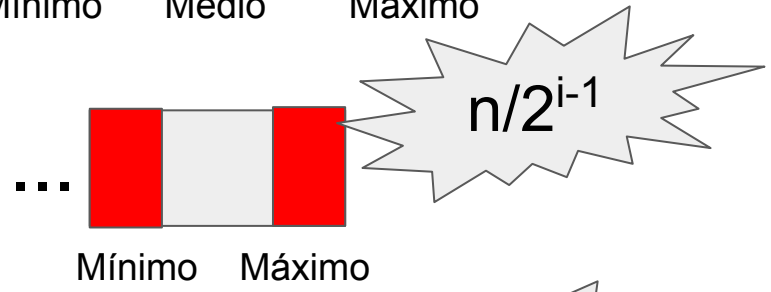
```
10          mid = (low + high)//2
```

```
17          else: return buscaBinaria(L, e, low, mid - 1)
```

```
18          else:
19              return buscaBinaria(L, e, mid + 1, high)
```







Busca Binária

- Quando a Busca termina?
  - Seja i o número de passos até chegar ao passo básico da recursão.
  - Temos que $n/2^i=1 \Rightarrow n=2^i \Rightarrow \log_2 n=\log_2 2^i \Rightarrow i = \log_2 n$
- Logo, a complexidade do algoritmo recursivo é O(logn).

Busca Binária

- Lembre-se que $\log_y(x)$ indica o número de vezes que y precisa ser multiplicado para atingir x.

$$\log_2 8 = x \Leftrightarrow 2^x = 8 \Leftrightarrow x = 3, \quad 2 \times 2 \times 2 = 8$$

- Da mesma forma, dividir 8 por 2 um total de $\log_y(x)$ vezes resulta em 1.
- Logo, o termo **high–low** é cortado pela metade pelo menos $\log_2$(**high–low)** vezes antes de atingir 1.

Algoritmos de Ordenação

- Considerando que vamos executar uma busca binária, qual seria o aumento da complexidade ao se ordenar uma lista antes de executar a busca?
  - Suponho que a complexidade de ordenação seja O(ordenar(L)), ainda indefinida matematicamente: **ordenar(L)=???**
  - A busca binária leva O(log(len(L)))
  - Teríamos O(ordenar(L) + log(len(L))) ≤ O(len(L))?
  - Resposta: Não é possível ordenar L em tempo sublinear.

Algoritmos de Ordenação

- Vamos mudar um pouco: suponha agora que vamos executar a busca binária m vezes sobre a mesma lista L.
- A complexidade estimada mudaria para:
    - O(ordenar(L) + m*log(len(L))) ≤ O(m*len(L))?
    - Resposta: Se m cresce, O(ordenar(L)) se torna irrelevante.
- O valor de m dependerá de O(ordenar(L)):
    - Se O(ordenar(L))=O($2^{len(L)}$), precisaremos de um elevado valor de m.
- Felizmente, existem eficientes algoritmos de ordenação.

Algoritmos de Ordenação

- **Algoritmo selection sort**: simples mas **ineficiente!!!**

```
1   def selectionSort(L):
2     """Entrada: lista L contem elementos que
3     podem ser comparados usando o relacinonal >.
4     Saída: lista L ordenada em ordem crescente"""
5     suffixStart = 0
6     while suffixStart != len(L):
7       for i in range(suffixStart, len(L)):
8         if L[i] < L[suffixStart]:
9           #troca posição
10          L[suffixStart], L[i] = L[i], L[suffixStart]
11      suffixStart += 1
```

Algoritmos de Ordenação

- Invariante de laço:
  - Separa a lista L em parte 1, L[0:i], e parte 2, L[i+1:len(L)].
  - Ordena a parte 1 tal que nenhum de seus elementos seja maior que o menor elemento na parte 2.

```
[27, 55, 59, 63, 29, 29, 74, 27, 74, 62]
      i
[27, 27, 59, 63, 55, 29, 74, 29, 74, 62]
          i
[27, 27, 29, 63, 59, 55, 74, 29, 74, 62]
              i
[27, 27, 29, 29, 63, 59, 74, 55, 74, 62]
                  i
[27, 27, 29, 29, 55, 63, 74, 59, 74, 62]
                      i
[27, 27, 29, 29, 55, 59, 74, 63, 74, 62]
                           i
[27, 27, 29, 29, 55, 59, 62, 74, 74, 63]
                               i
[27, 27, 29, 29, 55, 59, 62, 63, 74, 74]
```

Algoritmos de Ordenação

- Prova da propriedade de loop invariante:
    - **Caso Base:**
        - prefix = [ ] e suffix=L. A propriedade de loop invariante é verdadeira de forma trivial.
    - Passo indutivo: O algoritmo desloca um elemento de suffix para prefix, adicionando o menor elemento de suffix no final de prefix, mantendo suffix ordenado.
        - Logo, a cada passo do laço, move-se um elemento de suffix para prefix.
        - Como no caso base, a propriedade é verdadeira, adicionando o menor elemento de suffix em prefix manterá a ordenação em prefix.
        - Além disso, como o menor elemento é removido de suffix, nenhum elemento em prefix será maior que o menor elemento de suffix.
    - Terminação:
        - Ao final do laço, prefix contém a lista inteira e suffix estará vazia. Logo, L estará ordenada em ordem crescente.

Algoritmos de Ordenação

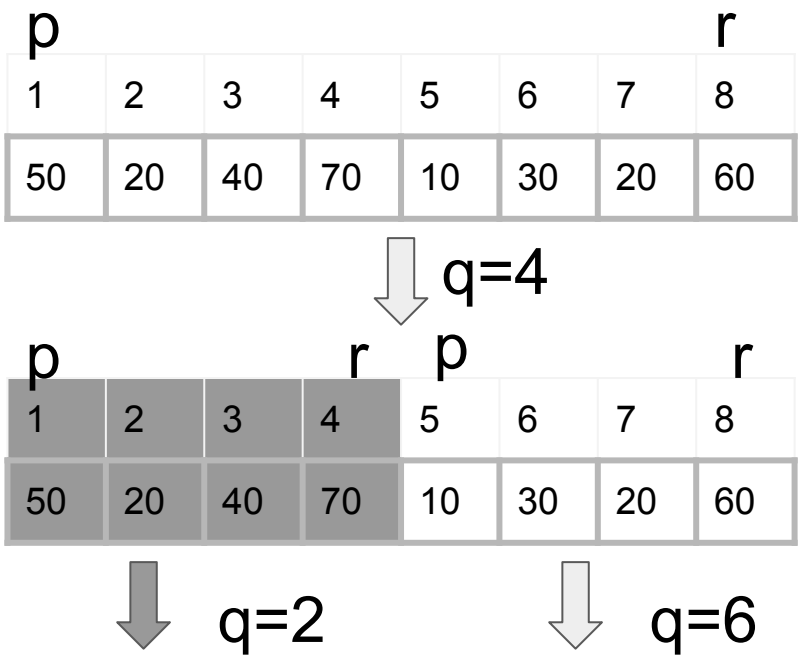
- **Complexidade:** $O(len(L)^2)$

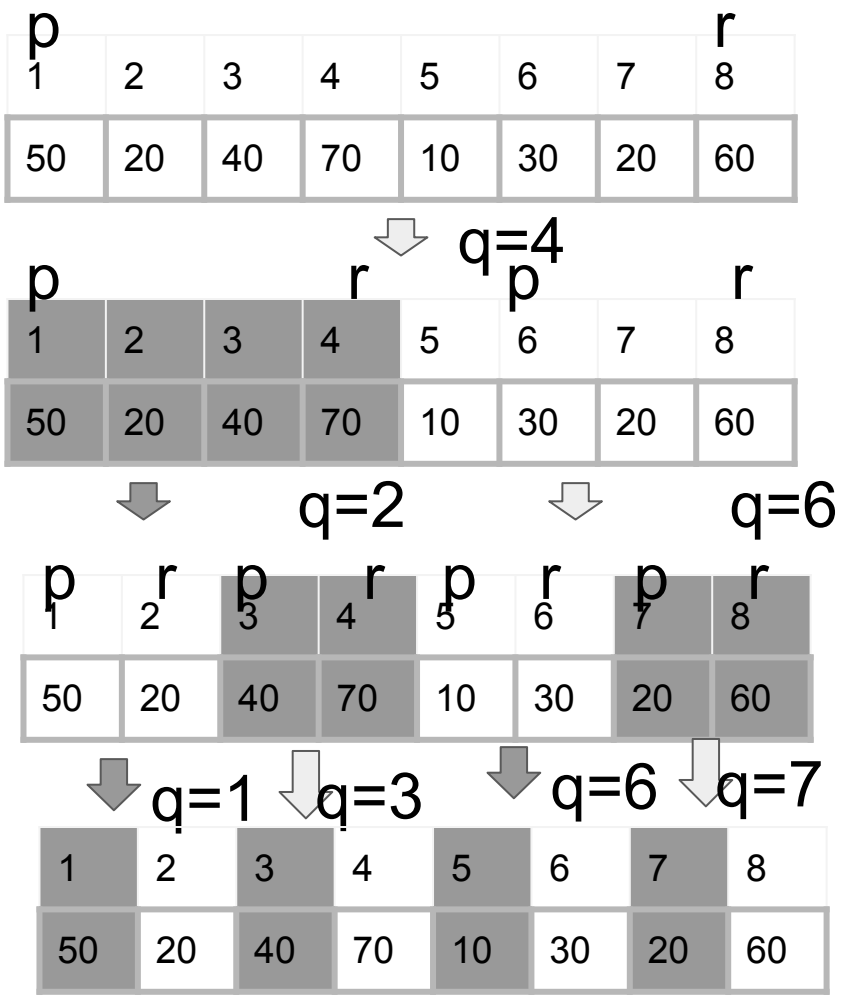
```
1   def selectionSort(L):
2     """Entrada: lista L contem elementos que
3     podem ser comparados usando o relacinonal >.
4     Saída: lista L ordenada em ordem crescente"""
5     suffixStart = 0
6     while suffixStart != len(L):
7       for i in range(suffixStart, len(L)):
8         if L[i] < L[suffixStart]:
9           #troca posição
10          L[suffixStart], L[i] = L[i], L[suffixStart]
11      suffixStart += 1
```

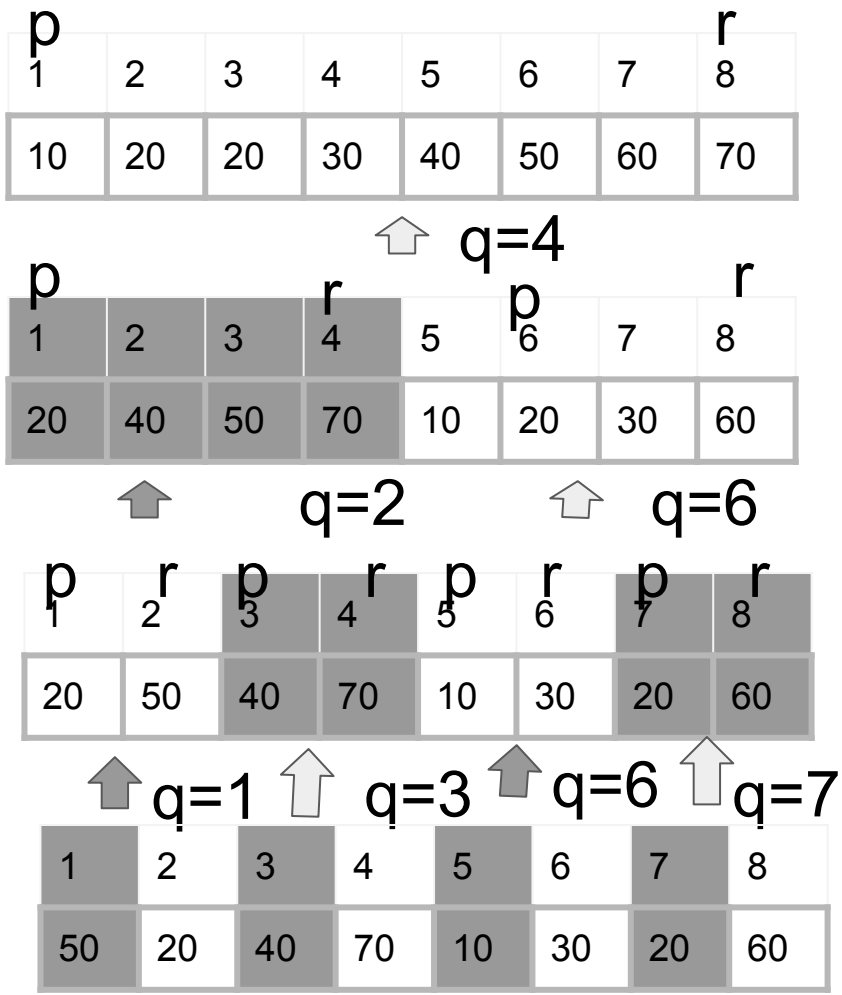
Algoritmo de Ordenação

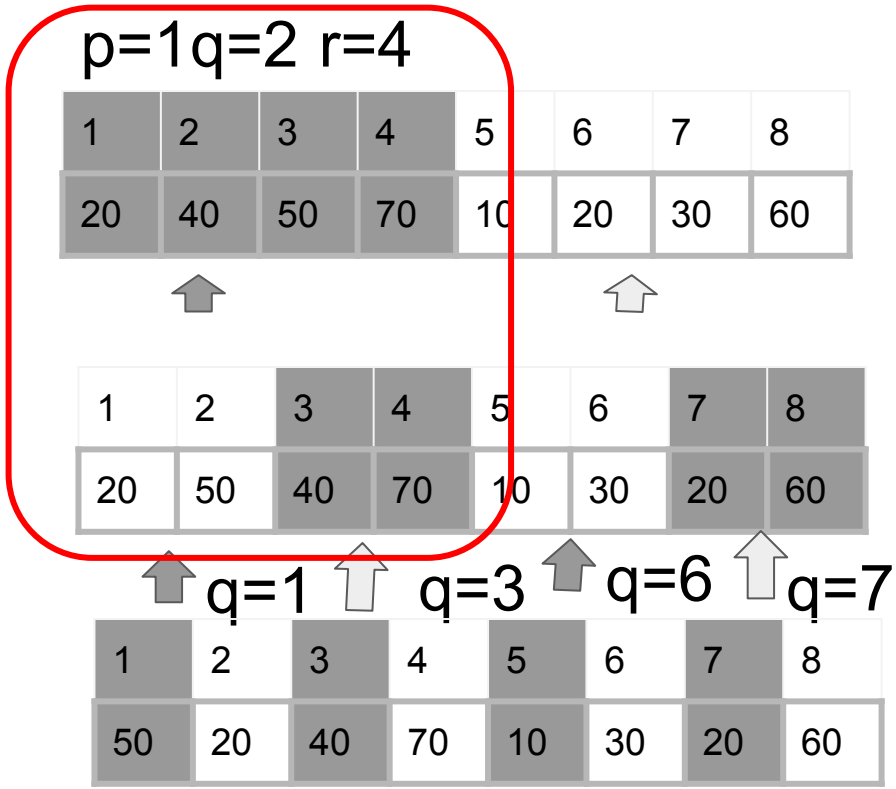
- Merge sort
  - Dá para fazer melhor que $O(len(L)^2)$!!
  - Aplica a abordagem **divisão e conquista.**
- Ideia da divisão e conquista:
  - Há um limiar (threshold) para até onde o tamanho da entrada pode ser subdividido
  - Há um tamanho e número de instâncias do subproblemas para o qual a instância original pode ser dividida
  - O algoritmo combina as soluções dos subproblemas para obter a solução do problema original.

Merge sort


p
r
1 2 3 4 5 6 7 8
50 20 40 70 10 30 20 60
q=4
p
r
p
r
1 2 3 4 5 6 7 8
50 20 40 70 10 30 20 60
q=2
q=6


p
r
1 2 3 4 5 6 7 8
50 20 40 70 10 30 20 60
q=4
p
r
p
r
1 2 3 4 5 6 7 8
50 20 40 70 10 30 20 60
q=2
q=6
p r p r p r p r
1 2 3 4 5 6 7 8
50 20 40 70 10 30 20 60
q=1
q=3
q=6
q=7
1 2 3 4 5 6 7 8
50 20 40 70 10 30 20 60


p
r
1 2 3 4 5 6 7 8
10 20 20 30 40 50 60 70
q=4
p
r
p
r
1 2 3 4 5 6 7 8
20 40 50 70 10 20 30 60
q=2
q=6
p r p r p r p r
1 2 3 4 5 6 7 8
20 50 40 70 10 30 20 60
q=1
q=3
q=6
q=7
1 2 3 4 5 6 7 8
50 20 40 70 10 30 20 60


p=1q=2 r=4
1 2 3 4 5 6 7 8
20 40 50 70 10 20 30 60
1 2 3 4 5 6 7 8
20 50 40 70 10 30 20 60
q=1
q=3
q=6
q=7
1 2 3 4 5 6 7 8
50 20 40 70 10 30 20 60

Algoritmos de Ordenação

- **Merge Sort**

```python
def merge(left, right, compare):
    """Entrada: Duas listas ordenadas
       Saída: Uma lista ordenada com elementos
       das duas listas iniciais."""
    result = []
    i,j = 0, 0
    while i < len(left) and j < len(right):
        if compare(left[i], right[j]):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
```

```python
def mergeSort(L, compare = operator.lt):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = mergeSort(L[:middle], compare)
        right = mergeSort(L[middle:], compare)
        merg = merge(left, right, compare)
        print(f'left:{left} right:{right} merge:{merg}')
        return merg
L = [random.randint(0,100) for i in range(10)]
print(L)
L=mergeSort(L)
print(L)
```

```
[53, 98, 52, 21, 15, 23, 38, 27, 72, 27]

left:[53] right:[98] merge:[53, 98]
left:[21] right:[15] merge:[15, 21]
left:[52] right:[15, 21] merge:[15, 21, 52]
left:[53, 98] right:[15, 21, 52]
                              merge:[15, 21, 52, 53, 98]
left:[23] right:[38] merge:[23, 38]
left:[72] right:[27] merge:[27, 72]
left:[27] right:[27, 72] merge:[27, 27, 72]
left:[23, 38] right:[27, 27, 72]
                              merge:[23, 27, 27, 38, 72]
left:[15, 21, 52, 53, 98] right:[23, 27, 27, 38, 72]
        merge:[15, 21, 23, 27, 27, 38, 52, 53, 72, 98]

[15, 21, 23, 27, 27, 38, 52, 53, 72, 98]
```

Algoritmo de Ordenação

- Complexidade do **merge()**:
  - Operações com tempo constante: comparação entre valores e cópia de elementos de uma lista para outra.
  - Número de comparações: O(len(L)), sendo L o tamanho da maior lista.
  - Número de cópias: O(len(L1) + len(L2)), cada elemento é copiado apenas uma vez.
  - Logo, merge () tem complexidade linear O(len(L)) em função do tamanho da maior entrada.

Algoritmo de Ordenação

- Complexidade do **mergeSort()**:
  - **merge()** tem complexidade O(len(L)).
  - O número total de elementos a serem combinados é len(L) dentro de cada nível de recursão.
  - Assim, a complexidade de **mergeSort()** é O(len(L)*número de níveis de recursão.
  - **mergeSort()** divide a lista pela metade a cada chamada, isso significa que o número de níveis de recursão será da ordem O((log(len(L)).
  - Logo, a complexidade do mergeSort é O(n*log(n)) para n= len(L).

Algoritmo de Ordenação

- **selectionSort()** é um **in-place sorting algorithm**, realizando troca de elementos dentro da lista.
- Logo, **selectionSort()** gasta apenas uma quantidade constante de memória extra para armazenamento.
- **mergeSort()** executa cópias da lista, levando a uma complexidade O(len(L)) em termos de espaço de armazenamento.
- Logo, **mergeSort()** pode ter problemas para listas extensas.

Tabela Hash

- Vimos que uma boa ideia para realizar busca em listas é associar merge sort com busca binária.
  - merge sort realiza a ordenação em O(n*log(n)) e podemos aplicar busca binária na lista com O(log(n)).
  - Para realizar k buscas na lista, teremos O(n*log(n) + k*log(n)).
- Ordem logarítmica é a melhor para executar uma busca quando um pré-processamento precisa ser feito?
  - Alternativa: Tabela Hash

Tabela Hash

- Idéia básica:
  - Converter uma chave de acesso para um inteiro e usar esse inteiro como um índice para acesso em uma lista.
  - Isso leva a um acesso que pode ser feito em tempo constante.
- Um compilador utiliza uma tabela de símbolos para relacionar símbolos aos dados associados.
  - Símbolos: nomes de variáveis, funções, etc..
  - Dados associados: localização na memória, gráfico de chamada, etc.
- Uma tabela de símbolos também é chamada de dicionário.

Tabela Hash

- Uma tabela de símbolos utiliza procedimentos para busca, inserção e exclusão.
- A tabela de símbolos não considera ordenação.
- Seja T uma tabela e um registro x com uma chave (símbolo) dado. Precisaremos implementar:
  - Insert (T, x)
  - Delete (T, x)
  - Search(T, x)
- As operações mencionadas ocorrem em O(1).

Tabela Hash

- Endereçamento direto
  - Suponha que todas as chaves (keys) são números naturais (grandes) num intervalo 0...m-1
  - As chaves são distintas e podemos definir um vetor T[0..m-1] tal que

    Se $x \in T$ && key[x] = i, então T[i] = x

    Caso contrário, T[i] = NULL

  - As operações (inserção, busca, exclusão) levam O(1)

Tabela Hash

- O endereçamento direto funciona quando o intervalo das chaves é pequeno.
- Exemplo: Considere chaves que são inteiros de 32-bit.
  - A tabela de endereçamento direto terá 4 bilhões de entradas.
  - Mesmo que a memória não seja um problema, o tempo para inicializar os elementos em NULL pode ser proibitivo.
- Alternativa: Transformação de Chave – Hashing
- Mapear chaves em um intervalo menor 0…m-1
- Desvantagem: colisões irão ocorrer.

Tabela Hash

- Mapeia o universo de chaves possíveis U para um conjunto {0,...,m-1} usando **funções hash**.

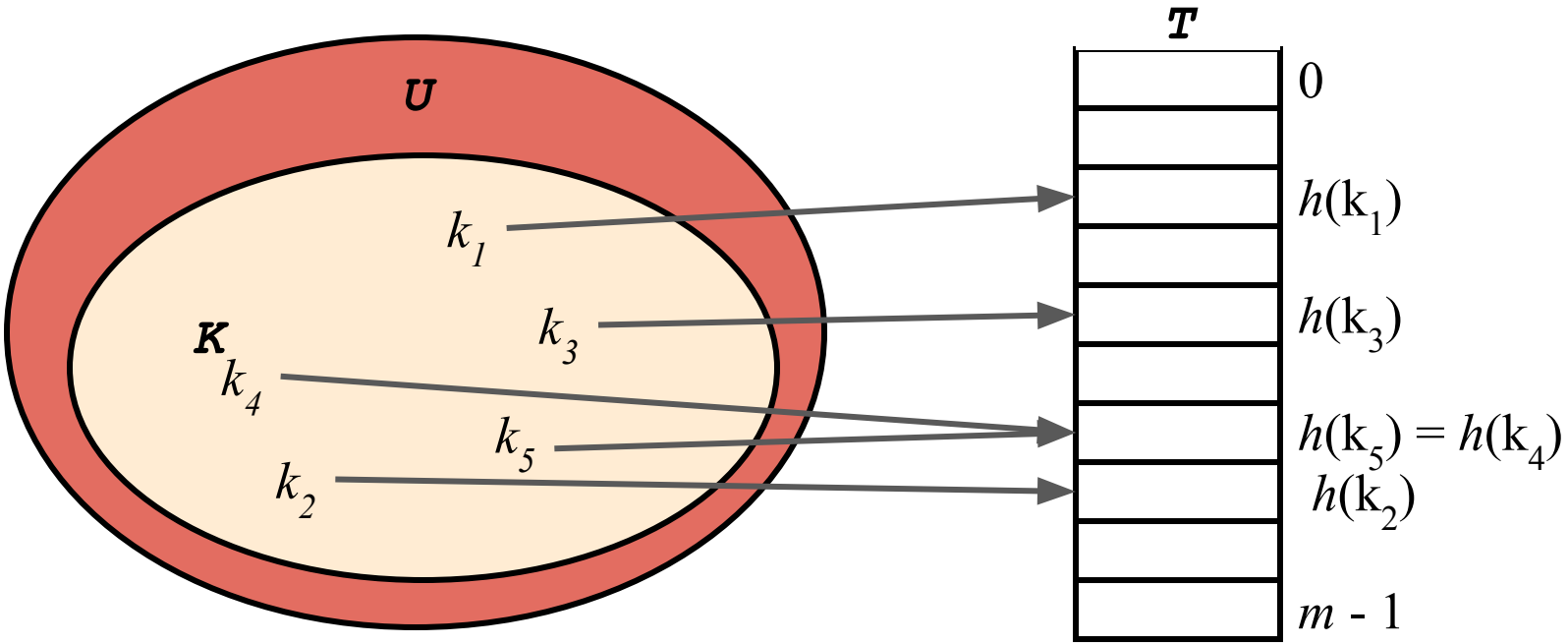


Tabela Hash

- **Função hash:** mapeia um espaço maior de entradas (todos os inteiros) para um espaço menor de saídas (inteiros entre 0 e 10000).
- Logo, pode ser usada para converter um vasto espaço de chaves para um espaço menor de valores inteiros usados como índices.
- Trata-se de um mapeamento muitos-para-um (many-to-one), ou seja, várias entradas podem ser mapeadas para a mesma saída.

Tabela Hash

- Colisões
  - U: Universo de chaves possíveis
  - K: chaves atuais

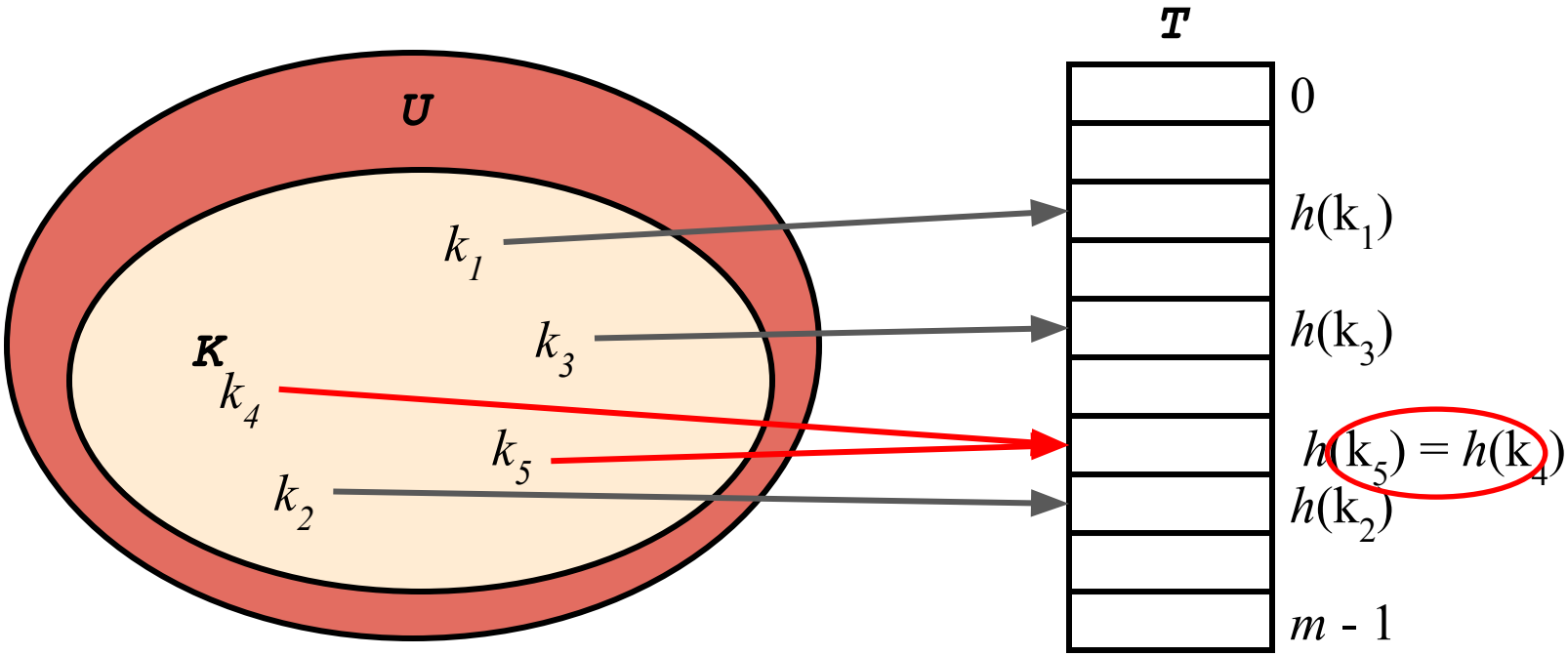


Tabela Hash

- **Colisão:** mapeamento de duas entradas para a mesma saída.
- Uma boa escolha de função hash leva a uma distribuição uniforme, onde cada saída tem a mesma probabilidade de ser obtida, minimizando a chance de colisões.

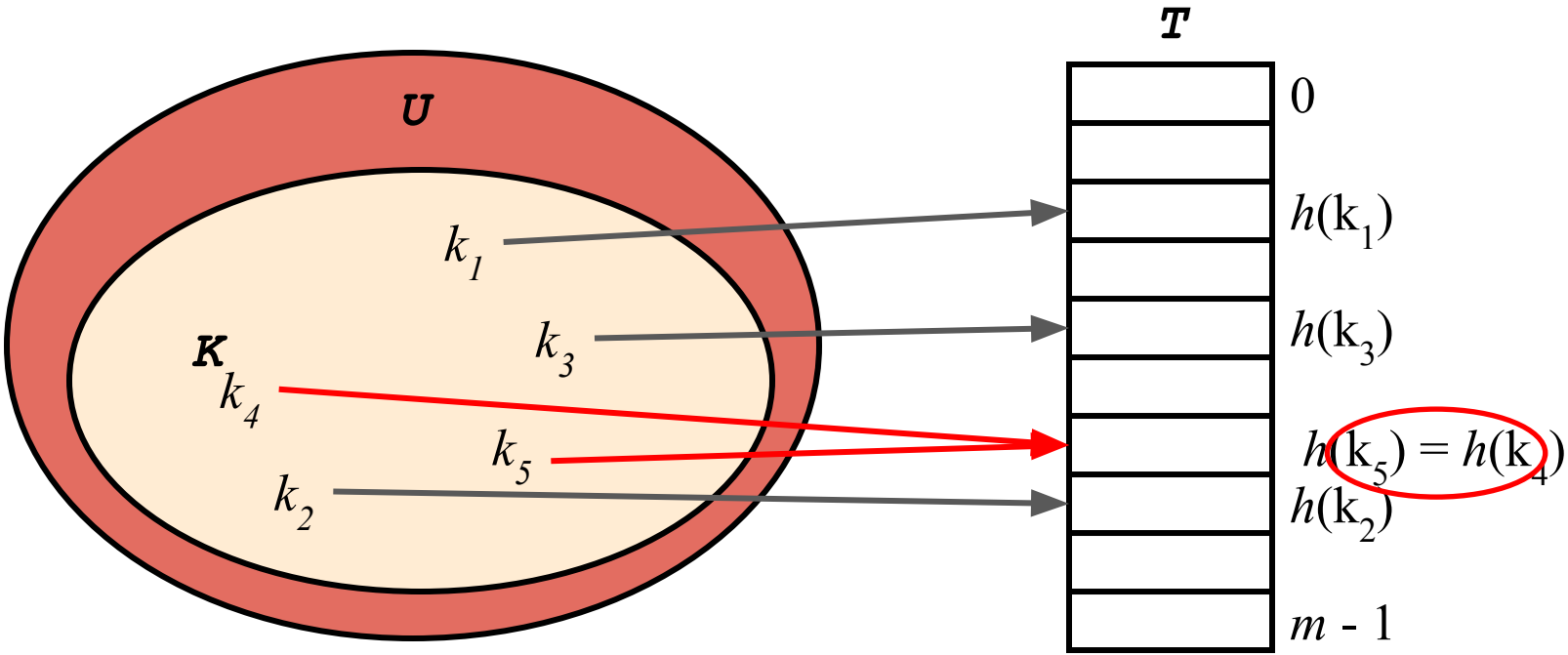


Tabela Hash

- Solução: Chaining
- Insere elementos atribuídos ao mesmo slot em uma lista ligada

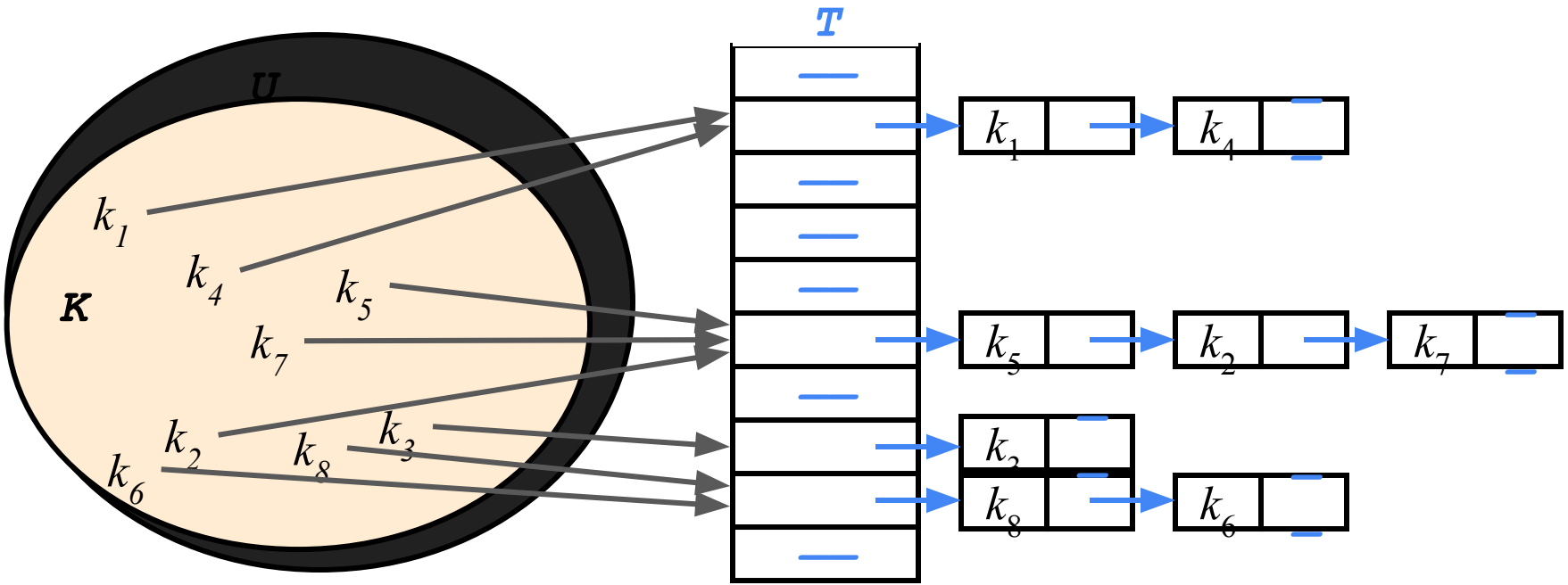


Tabela Hash

- Pior caso ocorre quando toda chave está atribuída ao mesmo slot
- Tempo de acesso Θ(n), se |K| =n

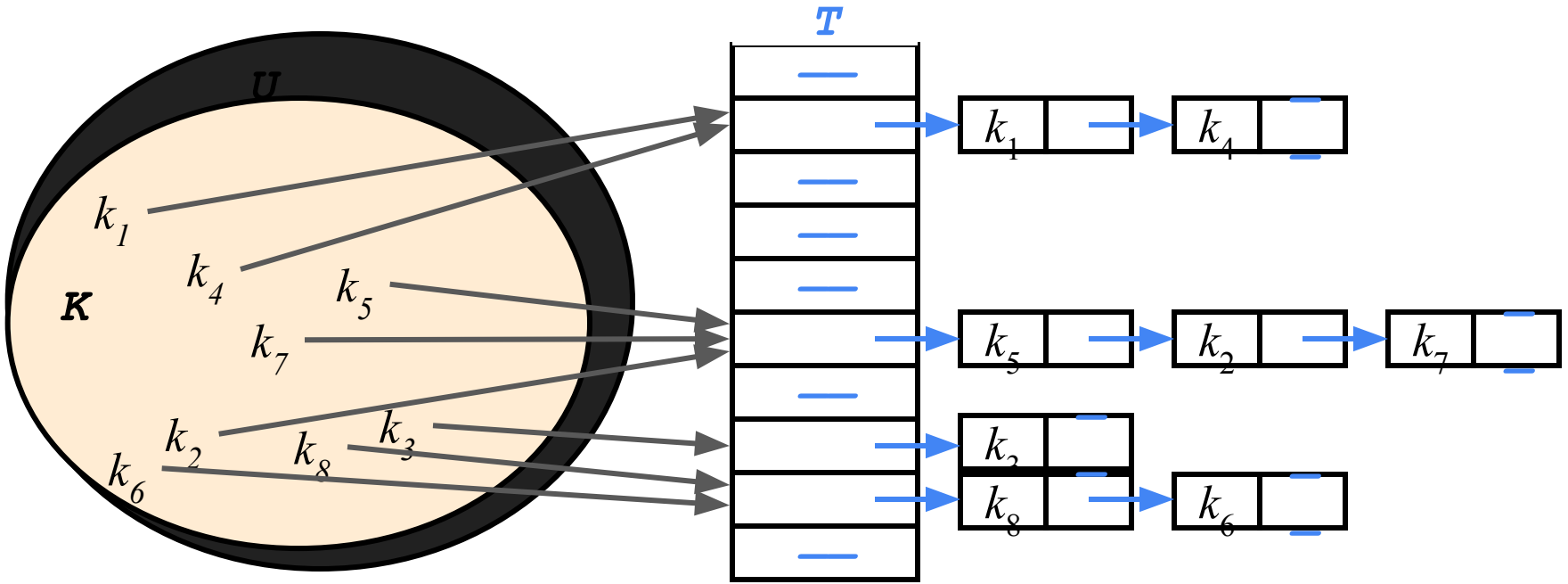


Tabela Hash

- Inserção em O(1), se o elemento não está em T
- Exclusão em O(1), se a lista é duplamente ligada e o elemento é dado.
- Busca com pior caso proporcional ao tamanho da lista.

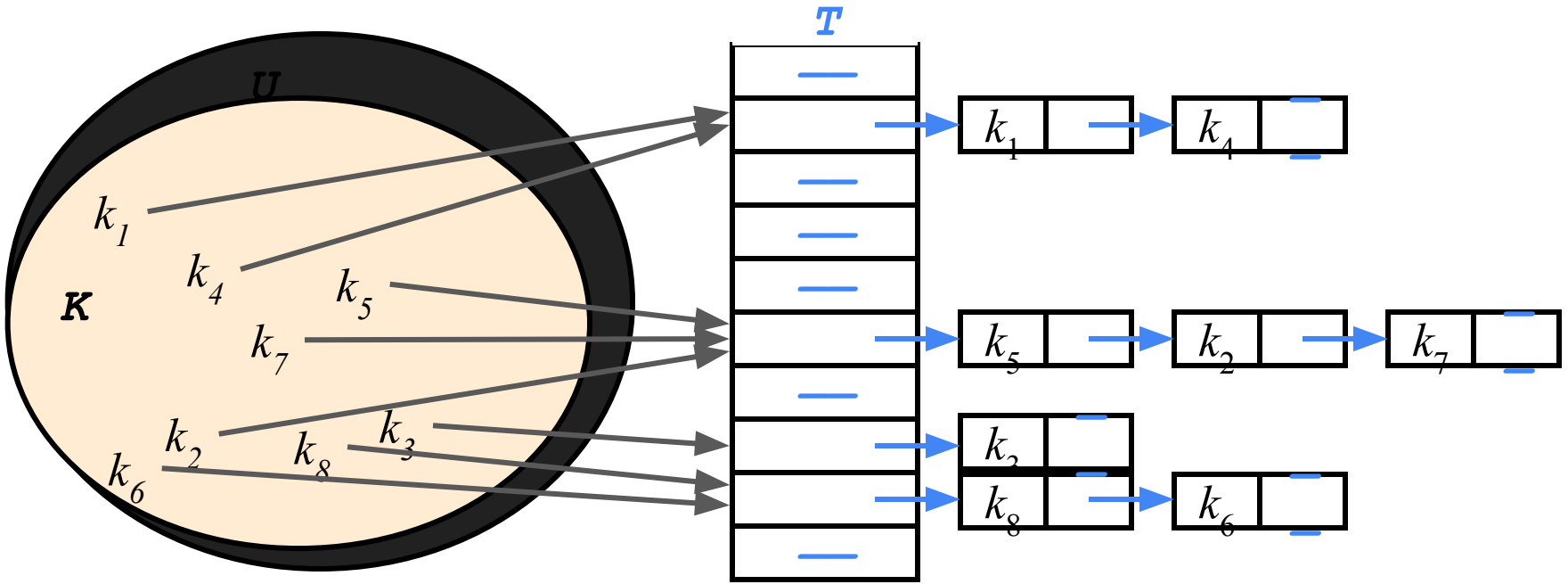


Tabela Hash

- Suponha que cada chave é igualmente provável de ser atribuída a qualquer posição na tabela T.
- Para n chaves e m entradas em T, temos o fator de carregamento (load factor) definido como: $\alpha = n/m$: número médio de chaves por entrada em T.
- O custo médio de uma busca sem sucesso por uma chave k será: $O(1+\alpha)$
- O(1): tempo para calcular h(k).
- $O(\alpha)$: tempo médio esperado para chegar ao final da lista na entrada T[h(k)].

Tabela Hash

- O custo médio de uma busca com sucesso por uma chave k será $O(1+\alpha)$
  - $O(1+ \alpha/2)=O(1+ \alpha)$
- Logo, o custo médio de uma busca (com ou sem sucesso) será $O(1+\alpha)$
- Se o número de chaves n é proporcional ao número de entradas em T , teremos
- $\alpha = n/m = O(m)/m=O(1)$ em média.

Tabela Hash

- A escolha de uma função hash adequada se torna fundamental:
  - Deve distribuir as chaves uniformemente nas entradas de T.
  - Não deve depender de padrões dos dados.
- Técnicas heurísticas podem ser utilizadas:
  - Método da divisão
  - Método da Multiplicação

Tabela Hash

- Método da divisão: h(k) = k mod m
- Assuma que todas as chaves são valores inteiros.
- O slot (entrada) da tabela é dado pelo valor do resto da divisão da chave k pela quantidade de slots m.
- Exemplo: m=12, k=100
    - h(100)=100 mod 12 = 4

Tabela Hash

- Desvantagem:
  - Não escolher um valor de *m* que tenha um pequeno divisor *d.*
  - Isso levará a uma maior incidência de chaves que são congruente módulo *d* , afetando a uniformidade desejada nas atribuições.
- Solução:
  - Escolher um tamanho de tabela com m= número primo.
  - Esse primo não pode estar próximo a uma potência de 2.

Tabela Hash

- Exemplo: chaves são caracteres com 8 bits num universo n=2000 caracteres (chaves).
- Se realizar uma busca sem sucesso em até 3 chaves não for um problema, podemos alocar uma tabela hash de tamanho m=701.
- m=701 é um primo próximo de 2000/3 sem estar próximo de uma potência de 2.
- h(k) = k mod 701.