UniAcademia

Disciplina

Estrutura de Dados

prof. Jacimar Tavares
jacimar.tavares@gmail.com

Módulo 04

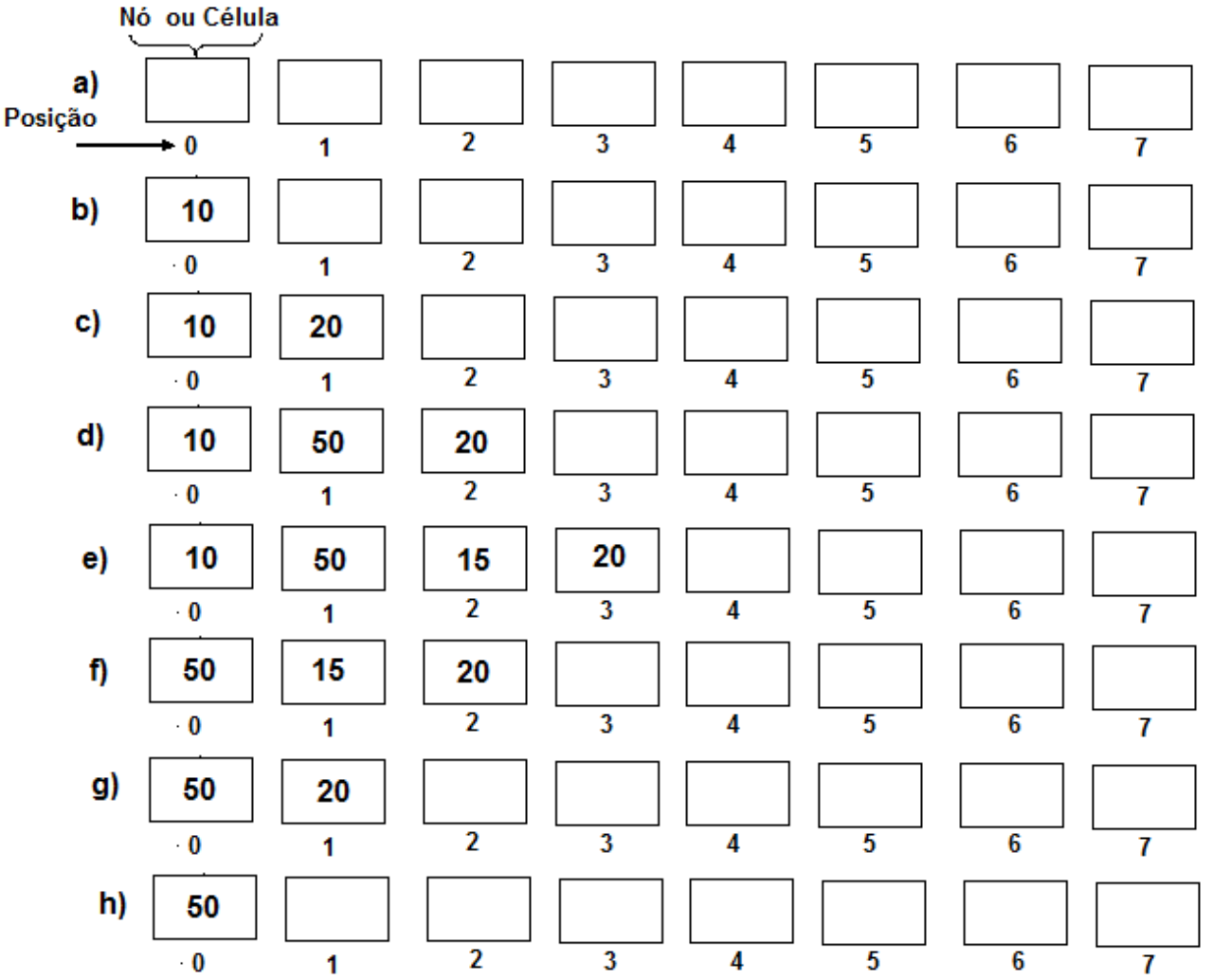
Fundamentos da Alocação de Memória e Ordenação

Alocação de memória

- **Overview**
  - Os termos estrutura de dados e tipo de dado concreto referem-se à representação interna de uma coleção de dados.
  - As duas estruturas mais usadas para implementar coleções em linguagens de programação são os arrays (alocação estática) e as estruturas ligadas (alocação dinâmica) .
  - Esses dois tipos de estruturas adotam abordagens diferentes para armazenar e acessar dados na memória do computador.

Alocação de memória

- **Overview**



Alocação de memória

- ***Lista como Tipo Abstrato de Dados***
  - A lista é a estrutura mais básica, das estruturas de dados existentes, ou seja,qualquer estrutura de dados pode ser “enxergada” como sendo uma estrutura lista. O que diferencia é a técnica de inserção, remoção e pesquisa dos seus elementos.
  - Existem alguns métodos básicos para o funcionamento de uma lista, o método insert(index,element) que insere o elemento na posição index e o método remove(index), que remove o elemento que está na posição “index”.

insert(index,value):	Insere o elemento “element” na posição index da lista.
remove(index):	Remove o elemento da posição index da lista. Este método retorna o elemento removido.

Alocação de memória

- ***Lista como Tipo Abstrato de Dados***
  –

Tabela 4.1 – Sequência de inserção e remoção numa Lista.

OPERAÇÃO	SAÍDA	CONTEÚDO DA LISTA
insert(0,5)	-	(5)
insert(1,10)	-	(5,10)
insert(2,4)	-	(5,10,4)
insert(0,15)	-	(15,5,10,4)
remove(2)	10	(15,5,4)
insert(1,33)	-	(15,33,5,4)
remove(0)	15	(33,5,4)

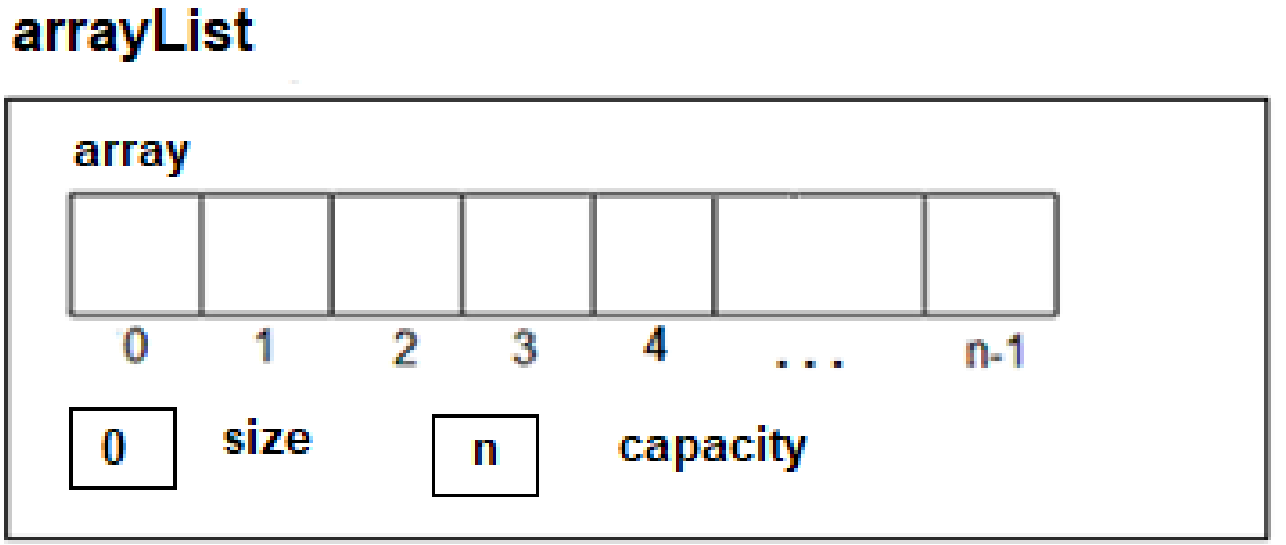
Alocação de memória

- ***Lista como Tipo Abstrato de Dados***
  - *Outros métodos possíveis*

__str_:	transforma a lista em um string
__len__:	retorna o tamanho da lista
__iter__:	retorna a lista como um interable.
__contains__:	Utilizado na sintaxe <elemento in list>. Retorna True caso exista e, False caso contrário.
__getitem__(index):	Retorna o element que está na posição index. Utilizado na sintaxe <list[index]>.
__setitem__(index):	Altera o elemento que está na posição index. Utilizado na sintaxe <list[index]=element>..

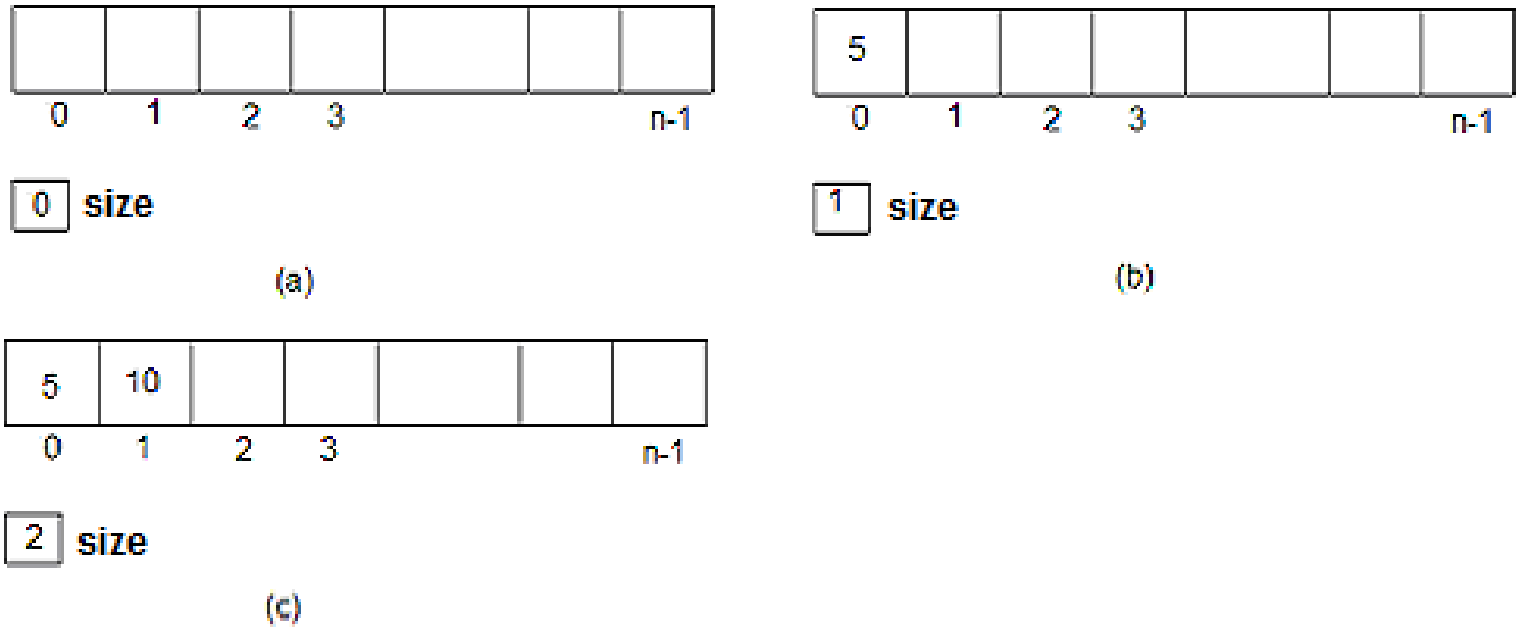
Alocação de memória

- ***Lista por contiguidade física***
  - Elementos estão em células adjacentes. A estrutura de dados utilizada para o armazenamento dos seus elementos, é o vetor (array).
    - Em Python, podemos considerar um vetor como sendo do tipo list.



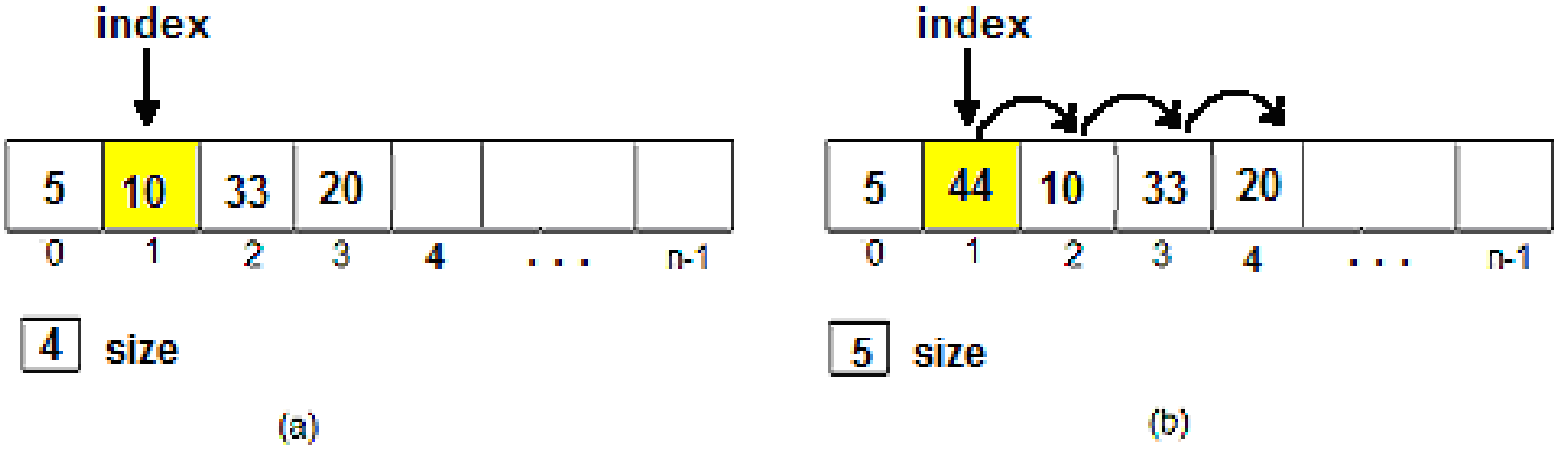
Alocação de memória

- ***Lista por contiguidade física***
  - Esquemática array



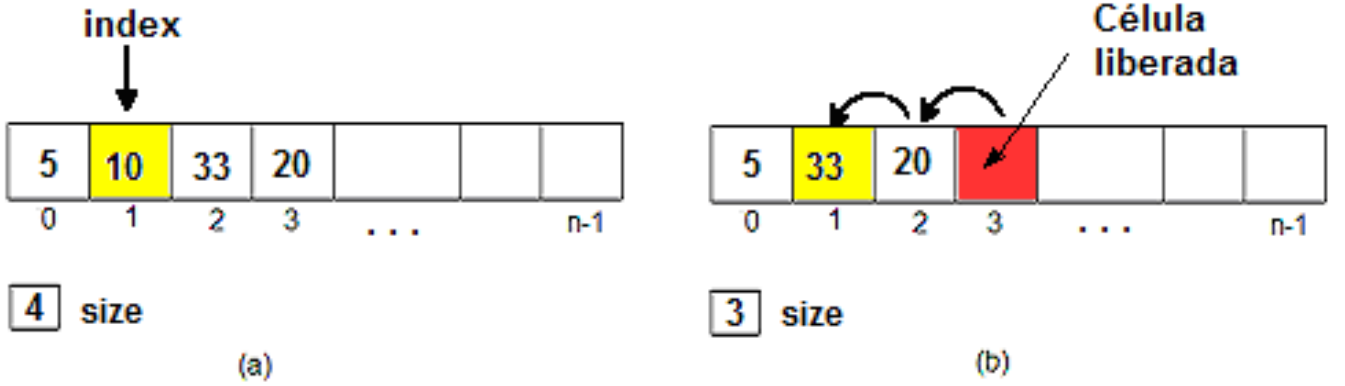
Alocação de memória

- ***Lista por contiguidade física***
  - A visão esquemática da ***inserção*** de um elemento numa posição "i", informada como parâmetro. Inicialmente, pode-se observar que todos os elementos que estão à direita da posição "i", devem ser ajustados uma posição à esquerda, a partir do último elemento, abrindo assim a "vaga" na posição solicitada para posterior inserção.



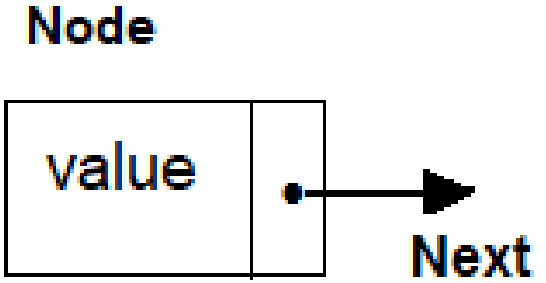
Alocação de memória

- ***Lista por contiguidade física***
  - A visão esquemática da **remoção** da posição “index”, informada como parâmetro (remove(index)).
  - A letra (a) representa uma **lista** contígua com 4 elementos. Na letra (b), é feito a remoção do elemento da posição do ponteiro “index”.
    - Neste caso as células à esquerda da posição “3” , correspondente, ao valor do campo size -1, até a posição “index”, são ajustadas uma casa à esquerda.



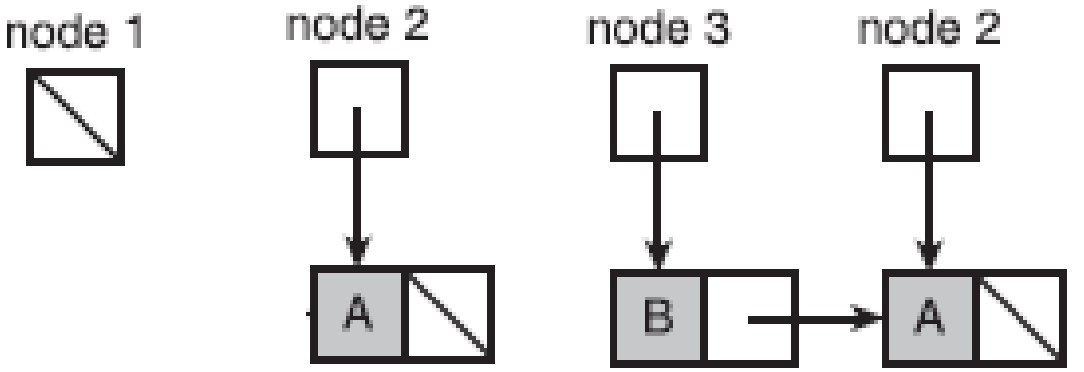
Alocação de memória

- ***A classe Node***
  - Uma outra forma de se implementar as listas lineares é a lista por encadeamento, isto é, as células são encadeadas por elos. Nesse caso, precisamos considerar as células como uma estrutura de dados. Esta estrutura denominada Node, contém, as informações do elemento (value) e um ponteiro, denominado de next, que será utilizado no encadeamento para apontar para a próxima célula da lista.
  - Ilustração da classe node, com a célula correspondente, com os dois elementos, o campo value, que guarda o elemento e, o campo next, que é um ponteiro para a próxima célula da lista.



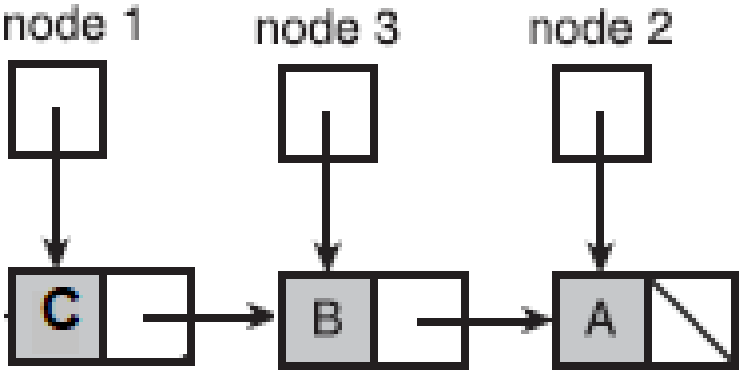
Alocação de memória

- ***A classe Node***
  - Estados das variáveis de instância da classe Node
  - Observe o seguinte:
    - node1 aponta para nenhum objeto de nó ainda (é None).
    - node2 e node3 apontam para objetos que estão vinculados.
    - node2 aponta para um objeto cujo próximo ponteiro é None.



Alocação de memória

- ***A classe Node***
  - Visão esquemática da inicialização da variável node1 inicializada com o ponteiro next = node3.

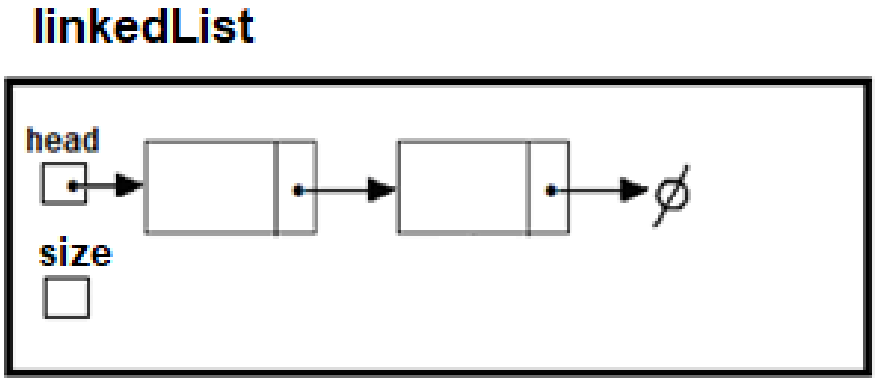


Alocação de memória

- ***A classe Node***
- ***Observações***
  - Um ponteiro, head(cabeça), gera a estrutura vinculada. Este ponteiro é manipulado de forma que o item inserido mais recentemente esteja sempre no início da estrutura.
  - Assim, quando os dados são exibidos, eles aparecem na ordem inversa de sua inserção. Além disso, quando os dados são exibidos, o ponteiro head (cabeça) é redefinido para o ponteiro next(próximo nó), até o ponteiro da cabeça torna-se None.
  - Assim, ao final desse processo, os nós são efetivamente excluídos da estrutura vinculada. Eles não estão mais disponíveis para o programa e são reciclados na próxima coleta de lixo (garbage collections).

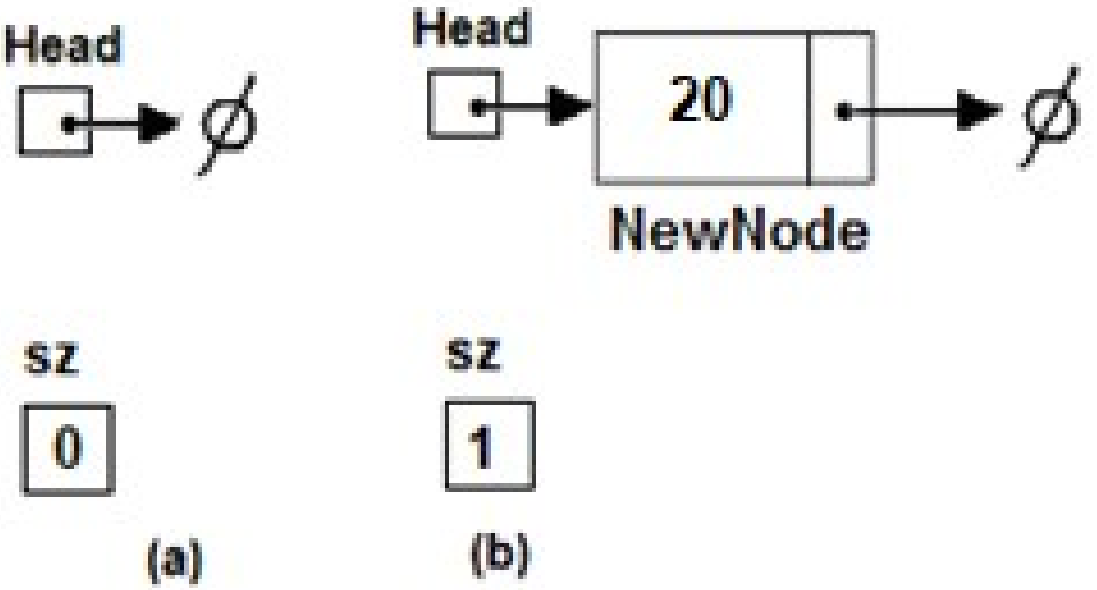
Alocação de memória

- ***Lista Encadeada***
  - Sequência de nós ligados pelos elos de cada nó (ponteiro next). Os nós que compõem a estrutura são da classe Node.
  - É necessário a criação de três elementos: o ponteiro head (cabeça da lista) que aponta para o primeiro nó da lista e, o campo size que armazena do tamanho da lista.
  - Na implementação o ponteiro head, na realidade é a primeira célula da lista.
  - Em tempo de inserção e remoção o campo size deverá ser atualizado.



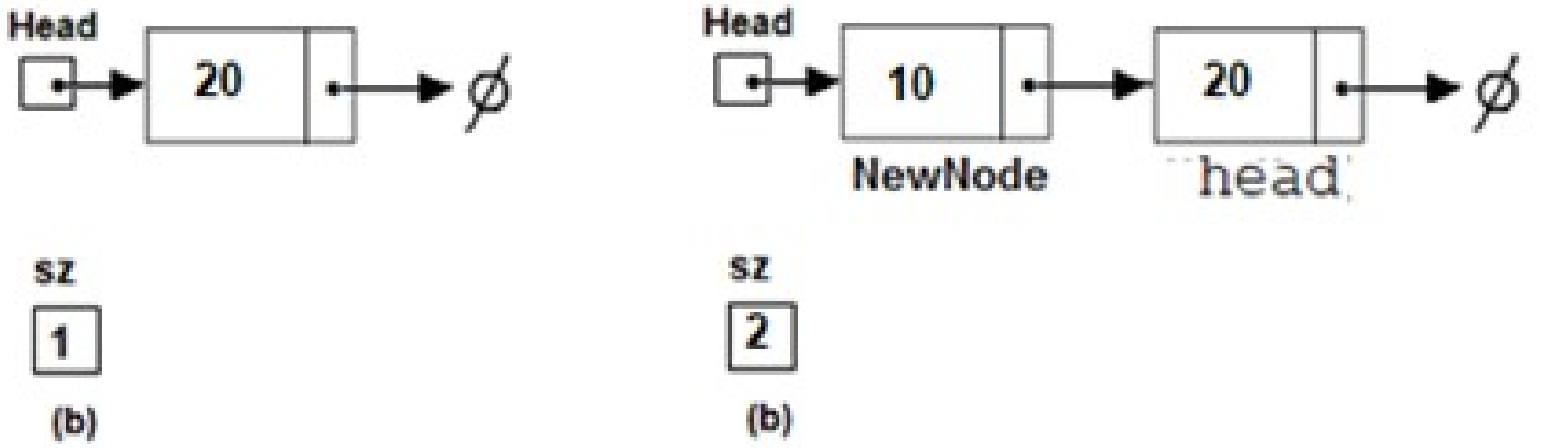
Alocação de memória

- ***Lista Encadeada***
  - Inserção



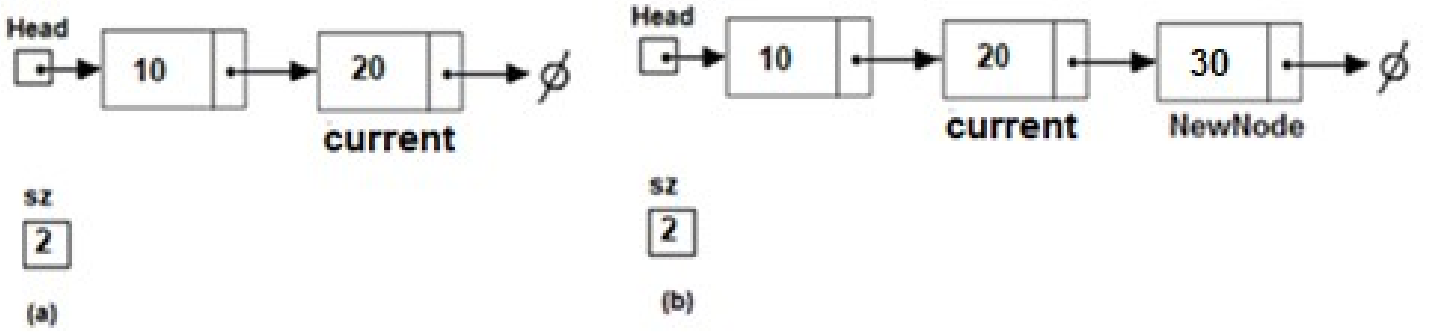
Alocação de memória

- ***Lista Encadeada***
  - **<u>Insere o primeiro</u>**, passando como parâmetro o valor = 10.
    - A letra a, corresponde ao estado antes da inserção e a letra b, após a inserção.



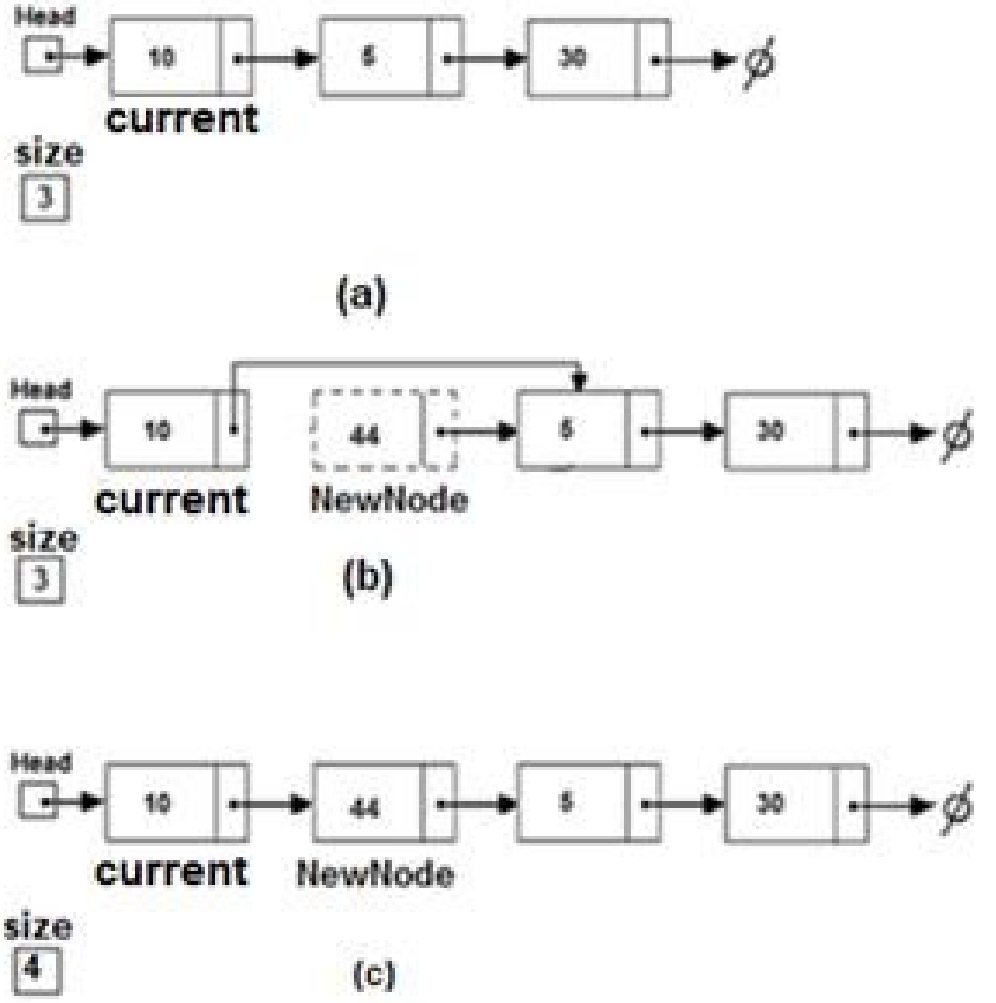
Alocação de memória

- ***Lista Encadeada***
  - **<u>Insere o último</u>**, Primeiramente inicializa-se a variável newNode com um nó inicializado com o valor a ser inserido. Depois é feito um teste se o ponteiro head é igual a None.
    - No caso afirmativo atualiza-se o ponteiro head com o newNode, ou seja o último é o primeiro elemento a ser inserido a lista.
    - Caso contrário, é feita uma varredura na lista até o último nó. Para isso, cria-se um ponteiro denominado de current, inicializado com o valor do ponteiro head e o loop é executado até que o ponteiro current.next seja igual a None.



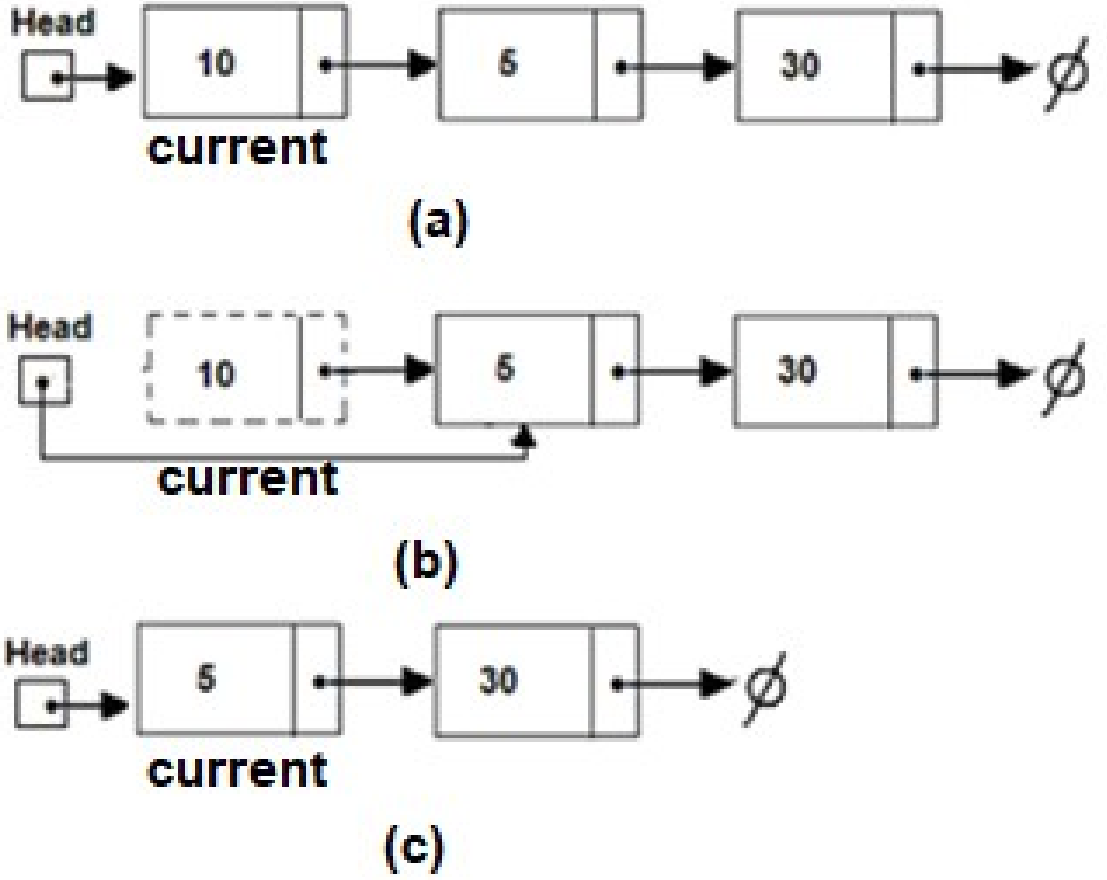
Alocação de memória

- *Lista Encadeada*
  - **Insere na posição index**



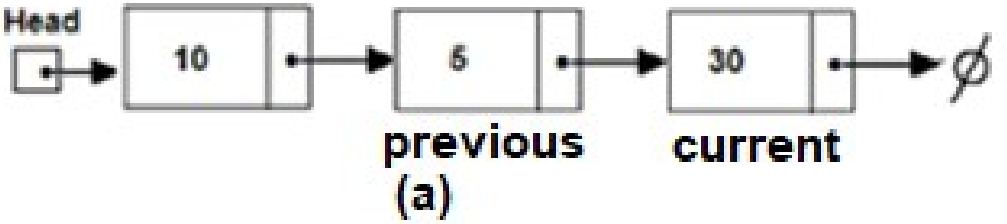
Alocação de memória

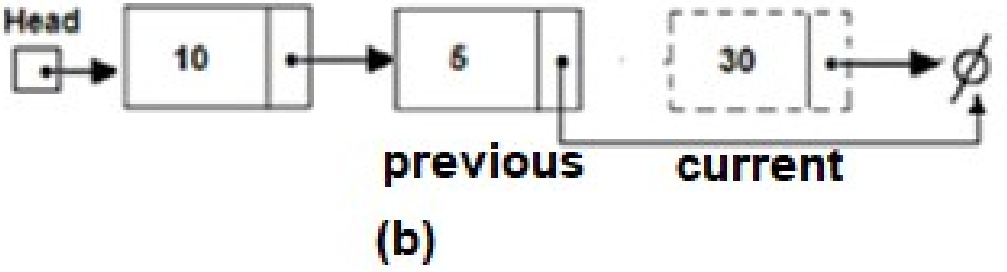
- *Lista Encadeada*
  - **Remove o primeiro**

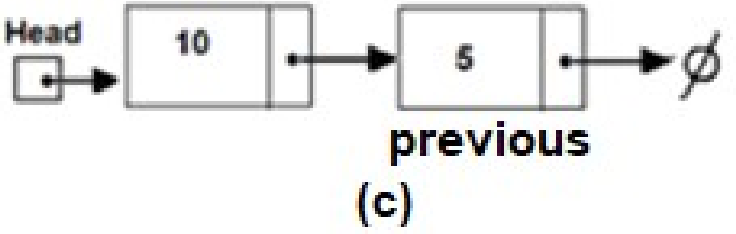


Alocação de memória

- *Lista Encadeada*
  - **Remove ultimo**

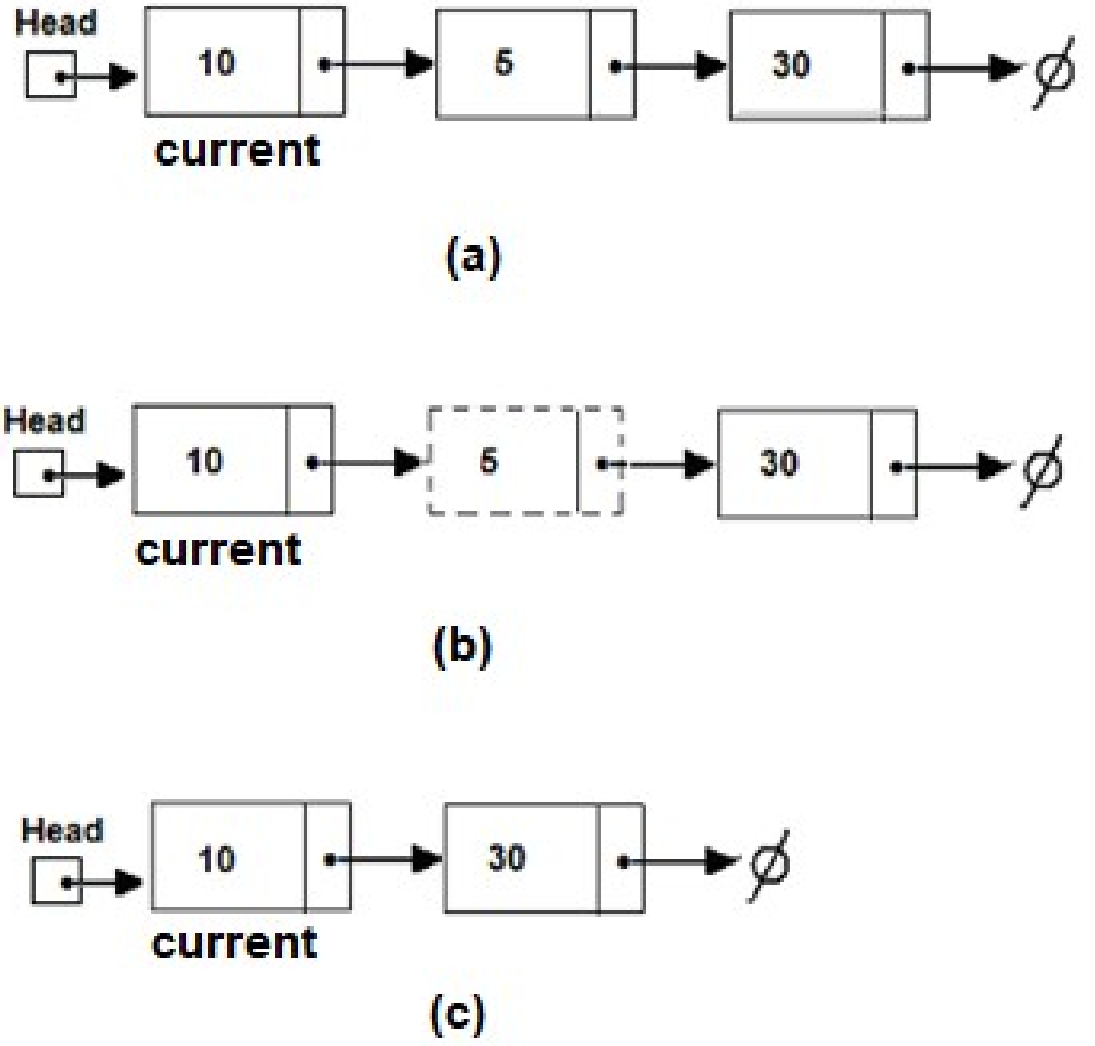






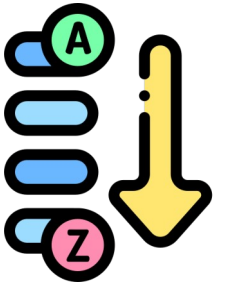
Alocação de memória

- ***Lista Encadeada***
  - **<u>Remove no index</u>**
    - Quando na lista encadeada se faz necessário descobrir o endereço da célula prévia em relação à um determinado node corrente, geralmente, é feito uma varredura em toda a lista para obtermos esse nodo prévio.



Fundamentos da Ordenação

- ***Ordenação***
  - Ordenar é o ato de rearranjar um certo conjunto de objetos em uma ordem ascendente ou descendente.
  - A grande motivação em se ordenar um arquivo está na procura de um determinado registro numa estrutura.
    - Por exemplo, considere um vetor **a**, como sendo um conjunto de chaves inteiras não organizadas sequencialmente.
    - Para se obter o endereço da célula cuja chave de pesquisa é igual à chave do registro deve-se utilizar uma função ***busca,*** *que* retorna o endereço da célula cuja chave é igual à chave de pesquisa ou “-1”, caso não exista chave igual à chave de pesquisa.



Fundamentos da Ordenação

- ***Ordenação: Busca em lista desordenada***

```
global NOT_FOUND
NOT_FOUND=-1
# iterative find
def find(alist,key):
    for i in range(0,len(alist)):
        if key == alist[i]:
            return i
    return NOT_FOUND
```

Fundamentos da Ordenação

- ***Ordenação: Busca em lista ordenada***

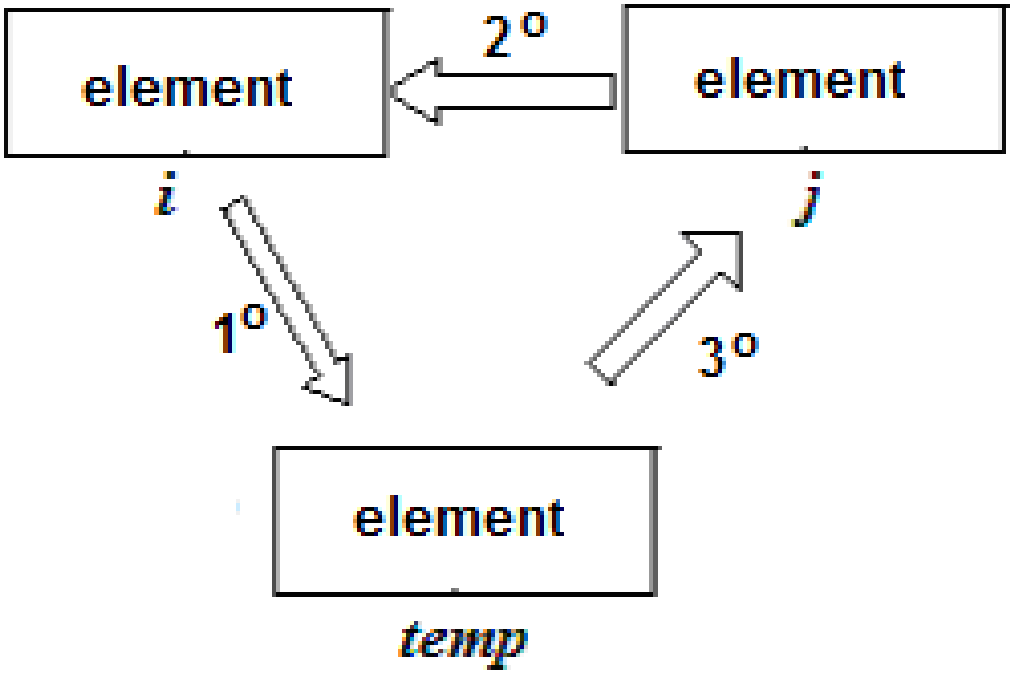
```
# iterative scan search
def scan(alist,key):
    for i in range(0,len(alist)):
        print(i)
        if key == alist[i]:
            return i
        if alist[i]>key:
            return NOT_FOUND
    return NOT_FOUND
```

Fundamentos da Ordenação

- ***Métodos de ordenação interna***
  - *Os métodos de ordenação interna são métodos de ordenação que utilizam arquivos lógicos implementados em memória principal. Em muito destes casos necessita-se de um método auxiliar na troca de posições dos elementos de um vetor.*
    - *Ordenação por Troca*
    - *Ordenação por Bolha (Bubble Sort)*
    - *Ordenação por Seleção (Select Sort)*
    - *Ordenação por Inserção*
    - *Ordenação Merge Sort*
    - *Ordenação Quick Sort*
    - *Ordenação Heap Sort*

Fundamentos da Ordenação

- ***Ordenação por Troca:***
  - Utilizam a ideia de se trocar um elemento que está numa determinada posição i, com um outro elemento que está numa posição j, tal que a chave do elemento da posição i seja menor que a chave do elemento da posição j.



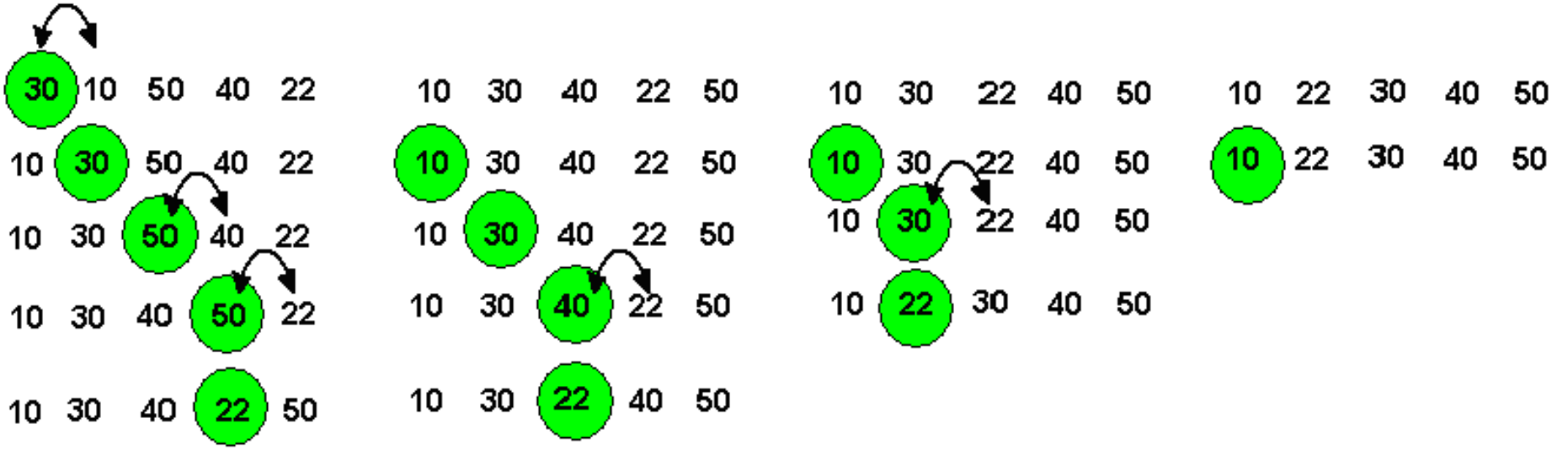
Fundamentos da Ordenação

- ***Ordenação por Troca:***
  - Utilizam a ideia de se trocar um elemento que está numa determinada posição i, com um outro elemento que está numa posição j, tal que a chave do elemento da posição i seja menor que a chave do elemento da posição j.

```
def swap(alist,i,j):
    temp = alist[i]
    alist[i] = alist[j]
    alist[j] = temp
```

Fundamentos da Ordenação

- ***Ordenação Bubble Sort***
  - O princípio básico consiste em se varrer o vetor várias vezes e, em cada passada, trocar-se cada elemento corrente (posição i) com o elemento seguinte (posição i + 1), caso c i > c i +1, onde ci e c i +1 são respectivamente as chaves dos registros corrente e o próximo.



Fundamentos da Ordenação

- ***Ordenação Bubble Sort***
  - Implementação

```
#Bubble Sort
def BubbleSort(alist):
    N = len(alist)
    for passnum in range(1,N):
        for i in range(0, N-passnum):
            if alist[i]> alist[i+1]:
                swap(alist, i, i+1)
    return alist
```

Fundamentos da Ordenação

- ***Ordenação Bubble Sort***
  - O fragmento ilustra o procedimento bubbleSort com a opção do corte sinalizando que não houve mais trocas.

```
def bubbleSort(alist):
    N = len(alist)
    for passnum in range(1,N):
        trocou=False
        for i in range(0,N-passnum):
            if alist[i]>alist[i+1]:
                sort.swap(alist,i,i+1)
                trocou=True
        if(not trocou):break
    return alist
```

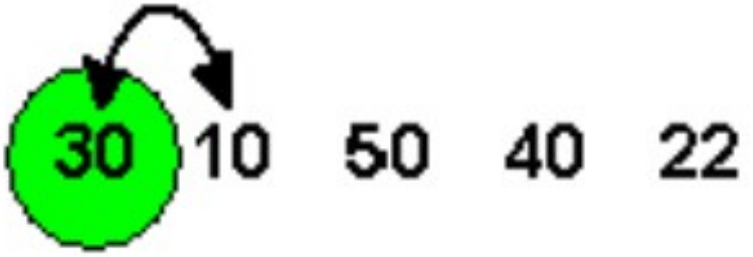
Atividades

- Implemente o método bubble sort para ordenar as listas abaixo:
  - A) [2,5,9,22,1]
  - B) [1,2,5,9,22]
  - C) [22,9,5,2,1]
  - D) [22,9,1,2,5]

- Faça prints para mostrar como as listas foram ordenadas.
- Responda as perguntas:
  1) Qual das listas usou todos os caminhos do algoritmo BubbleSort?
  2) Qual das listas menos usou o algoritmo para ordenação?

Fundamentos da Ordenação

- ***Ordenação Select Sort***
  - O princípio básico consiste em se escolher o menor valor da chave de um vetor e trocar este registro com o primeiro registro da sequência.
  - Ordenar o conjunto de chaves 30, 10, 50, 40 , 22 utilizando o método de seleção. As chaves em negrito correspondem a sequência de execução. Pode-se observar que, no vetor anterior, o menor valor da chave é 10. Sendo assim, troca-se com o primeiro elemento: 30.

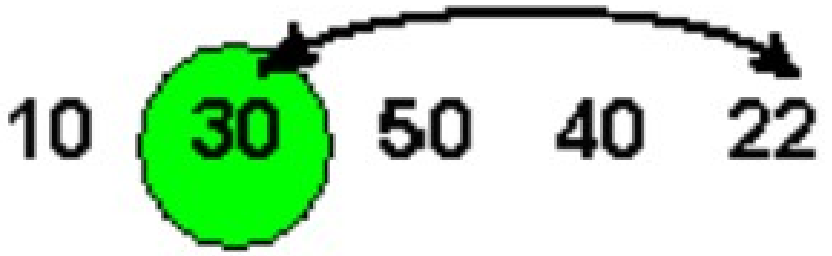


A nova sequência formada será :

10 30 50 40 22

Fundamentos da Ordenação

- ***Ordenação Select Sort***
  - O próximo registro é o segundo, chave igual a 30, e o menor valor a partir do segundo elemento é o de chave igual a 22. Troca-se o registro de chave igual a 22 com o segundo registro de chave igual a 30.



A nova sequência formada será :

10 22 50 40 30

Fundamentos da Ordenação

- ***Ordenação Select Sort***
  - O próximo registro é o quarto, chave 40, e o menor valor a partir dele, é ele mesmo. Logo, não efetua a troca. A sequência permanece inalterada.

Fundamentos da Ordenação

- ***Ordenação Select Sort***

```
def selectSort(alist):
   N = len(alist)
   for i in range(0,N-1):
      positionOfMin=i
      for j in range(i+1,N):
          if alist[j]<alist[positionOfMin]:
              positionOfMin = j
      sort.swap(alist,i,positionOfMin)
   return alist
```

Atividades

- Implemente o método select sort para ordenar as listas abaixo:
  - A) [2,5,9,22,1]
  - B) [1,2,5,9,22]
  - C) [22,9,5,2,1]
  - D) [22,9,1,2,5]

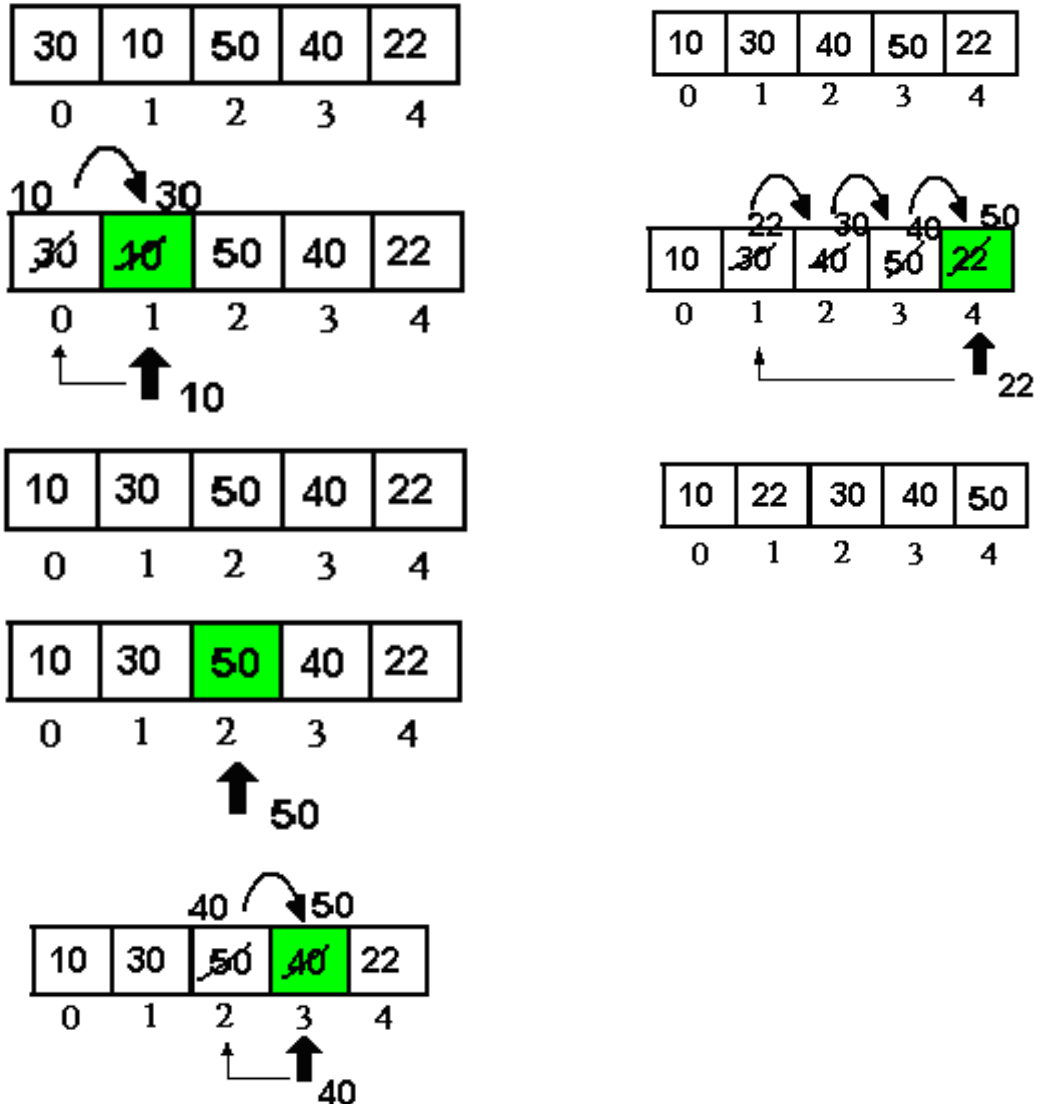
- Faça prints para mostrar como as listas foram ordenadas.
- Responda as perguntas:
  1) Qual das listas usou todos os caminhos do algoritmo Select Sort?
  2) Qual das listas menos usou o algoritmo para ordenação?

Fundamentos da Ordenação

- ***Ordenação por Inserção***
  - O conceito básico deste método consiste em inserir um registro R numa sequência de registros ordenados R1, R2, R3, ... Rn, de chaves C1 < C2 < C3 , .. < Cn, sendo que a nova sequência de n + 1 termos deverá ficar também ordenada.
  - A ideia básica está em varrer o vetor do segundo elemento até o fim e, para cada elemento, inseri-lo na lista formada com os elementos anteriores de tal forma que esta lista continue ordenada.

Fundamentos da Ordenação

- ***Ordenação por Inserção***



Fundamentos da Ordenação

- ***Ordenação por Inserção***

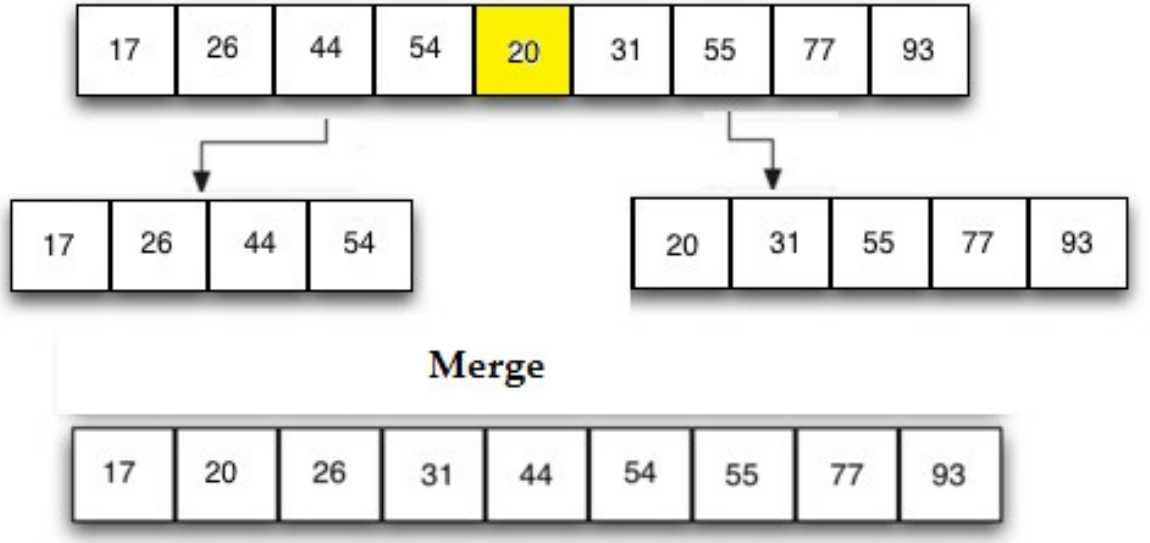
```
def insertSort(alist):
   N = len(alist)
   for i in range(1,N):
      aux=alist[i]
      j=i-1
      # Achar a posição do menor elemento
      while( j >= 0  and alist[j] >= aux):
            alist[j+1]=alist[j]
            j=j-1
      alist[j+1]=aux
   return alist
```

Atividades

- Responda:
  - Mostre como ficaria o vetor resultante enquanto:
    - I = 3, j=2
  - Mostre o valor de i e j quando aux == 22
  - Mostre o vetor resultante quando j= 0 pela segunda vez.

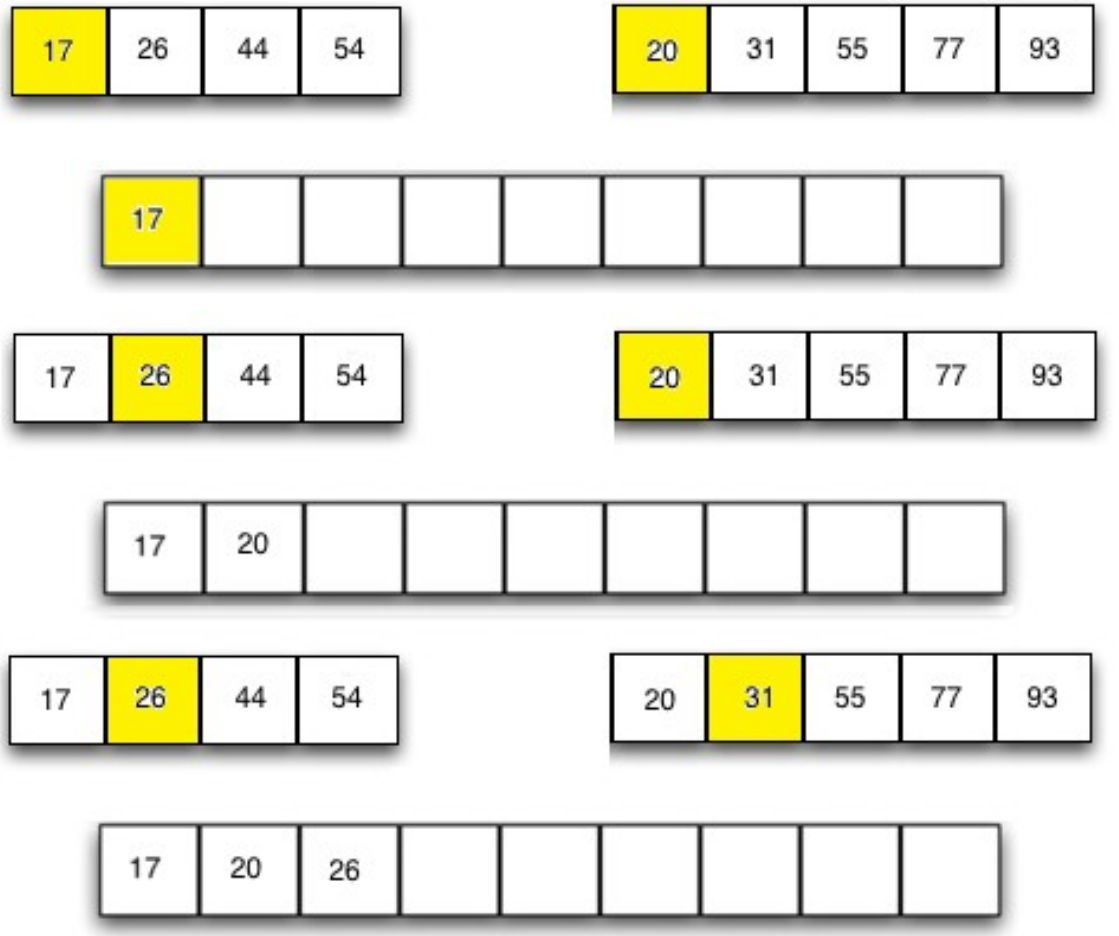
Fundamentos da Ordenação

- ***Ordenação com Merge Sort***
  - Este método consiste em intercalar dois vetores ordenados, gerando-se no final um novo vetor também ordenado
    - Considerando dois arquivos lógicos ordenados:
      - o primeiro do índice 0 até 3, e, o segundo do índice 4 até o final.



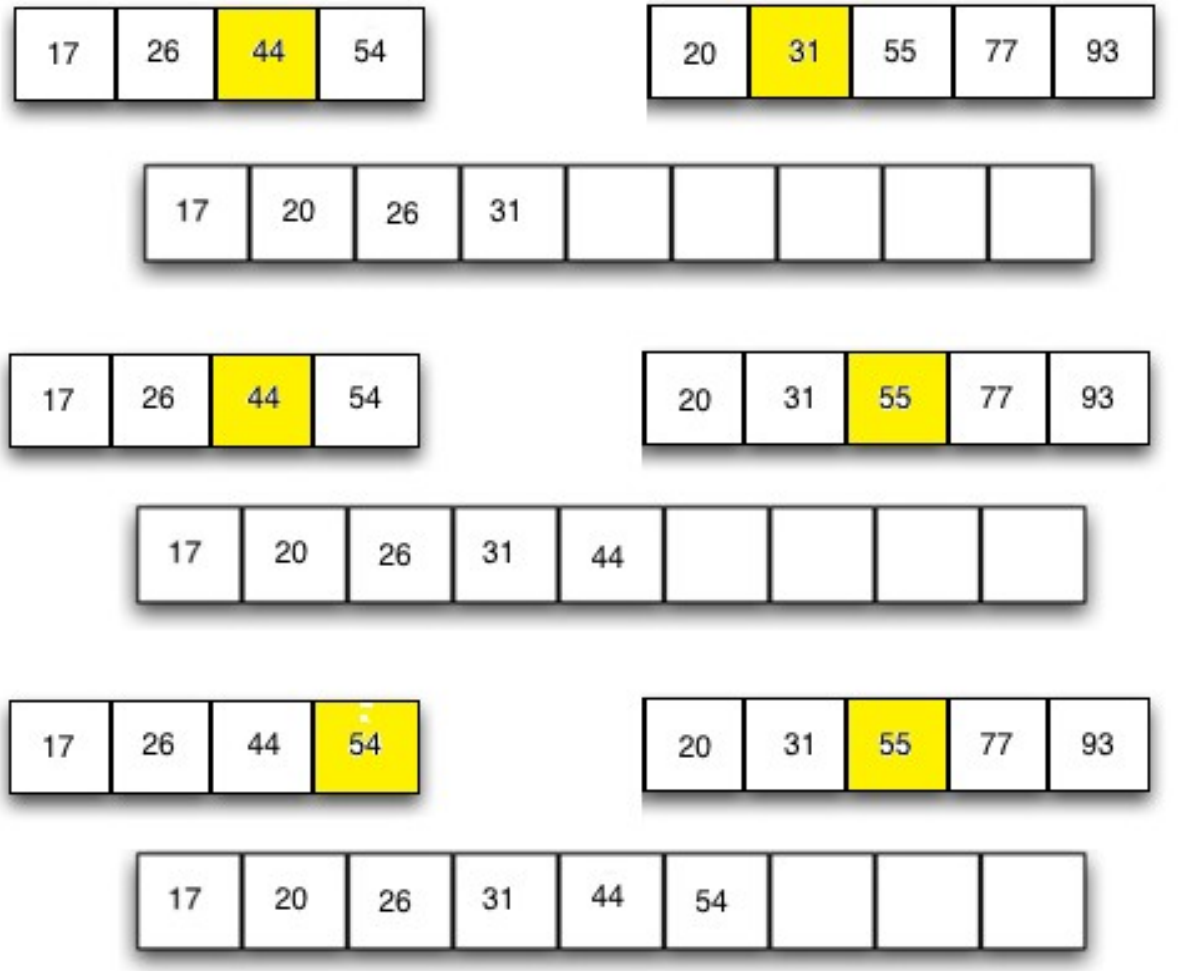
Fundamentos da Ordenação

- ***Ordenação com Merge Sort***
  - Exemplo.



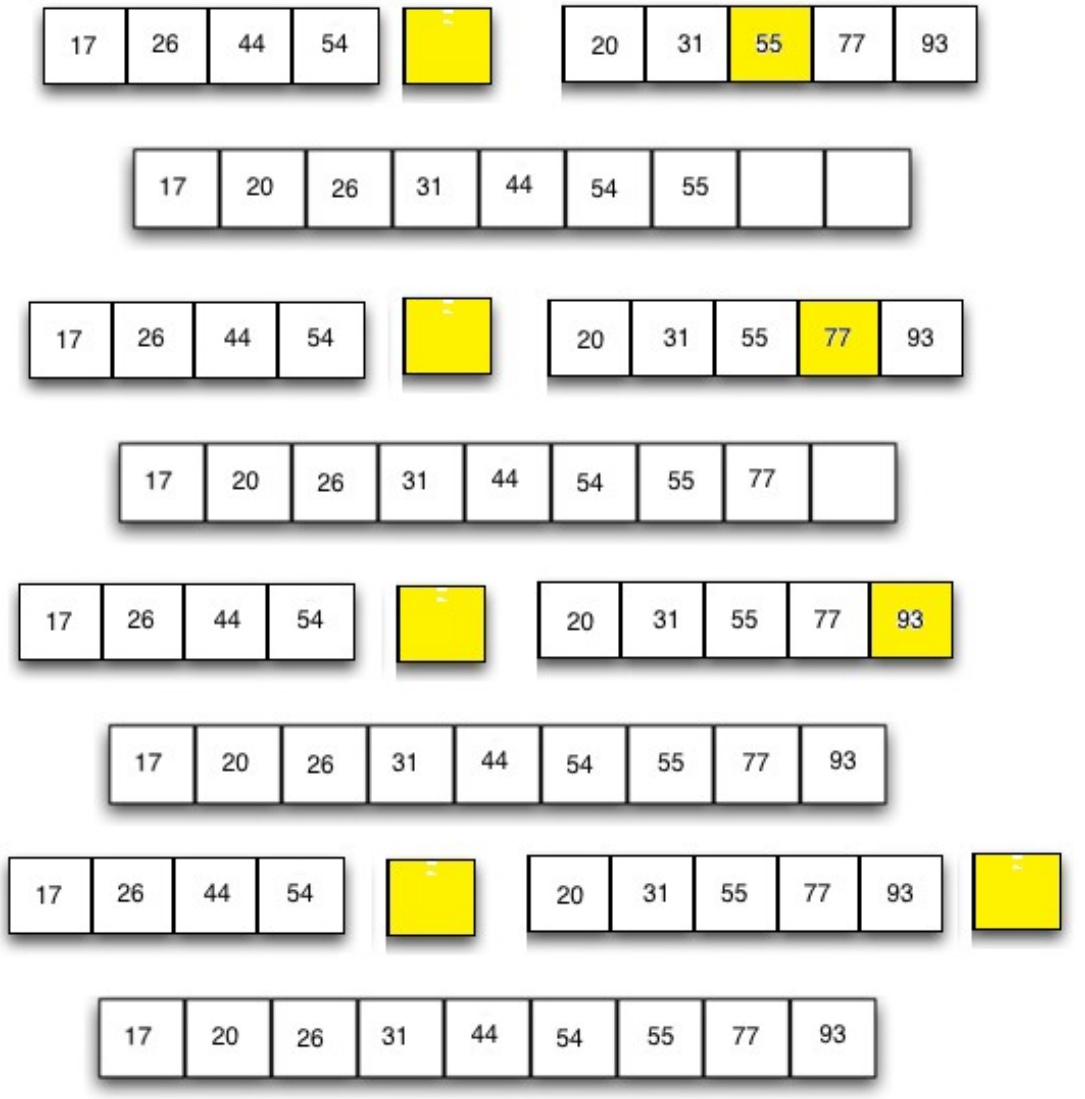
Fundamentos da Ordenação

- ***Ordenação com Merge Sort***
  - Exemplo.



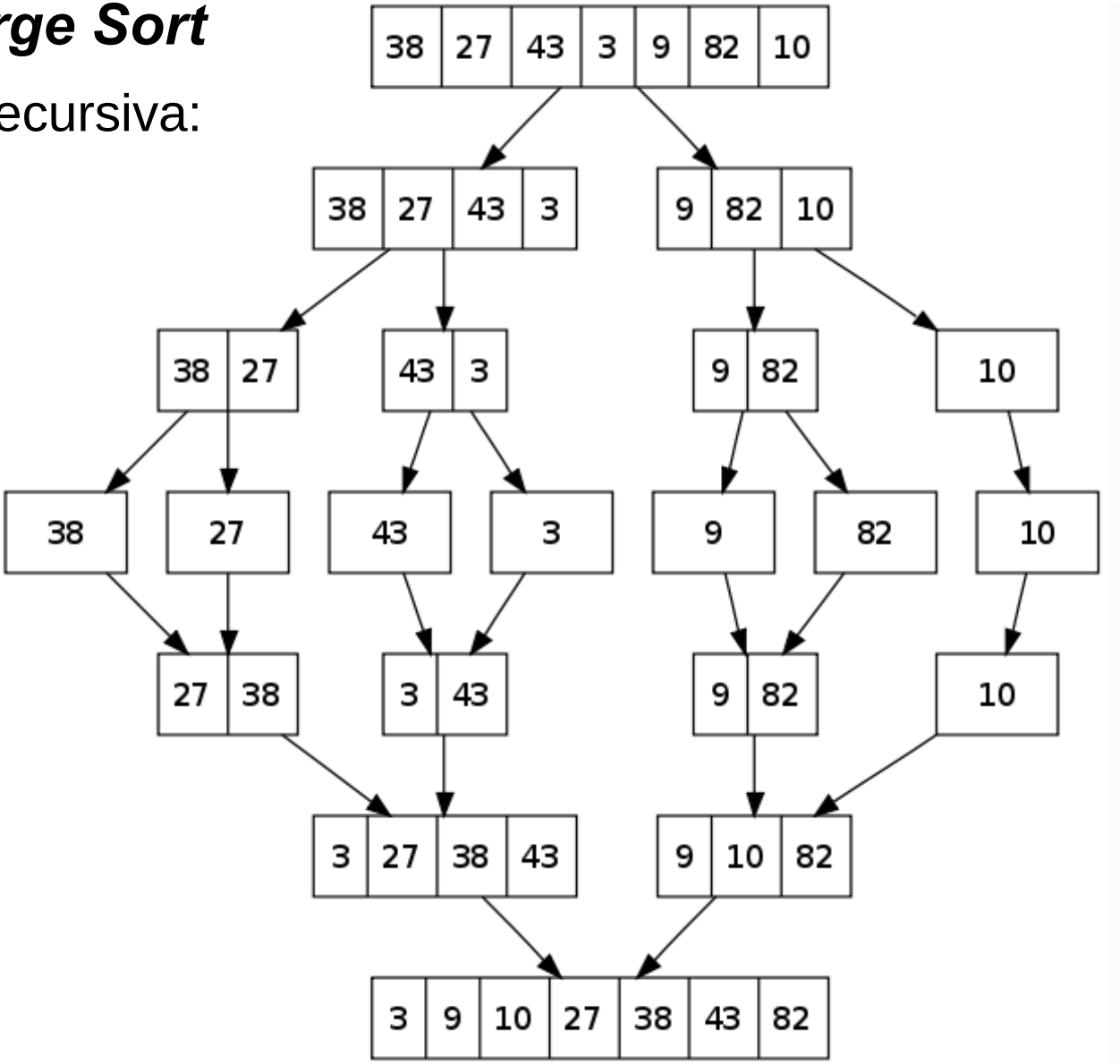
Fundamentos da Ordenação

- ***Ordenação com Merge Sort***
  - Exemplo.



Fundamentos da Ordenação

- ***Ordenação com Merge Sort***
  - Exemplo na forma recursiva:



Fundamentos da Ordenação

- ***Ordenação com Merge Sort***

```
# Merge de listas
def merge(alist, mid):
    ladoEsq = alist[:mid]
    ladoDir = alist[mid:]
    i=j=k=0
    while i < len(ladoEsq) and j < len(ladoDir):
        if ladoEsq[i] <= ladoDir[j]:
            alist[k]= ladoEsq[i]
            i+=1
        else:
            alist[k]=ladoDir[j]
            j +=1
        k +=1
    while i < len(ladoEsq):
        alist[k] = ladoEsq[i]
        i +=1
        k +=1
    while j < len(ladoDir):
        alist[k] = ladoDir[j]
        j +=1
        k +=1

    return alist
```

Fundamentos da Ordenação

- ***Ordenação com Merge Sort***

```
#Merge sort
def mergeSort(alist):
    N = len(alist)
    if len(alist)>1:
        mid = N // 2
        ladoEsq = alist[:mid]
        ladoDir = alist[mid:]
        ladoEsq = mergeSort(ladoEsq)
        ladoDir = mergeSort(ladoDir)
        alist = merge(ladoEsq+ladoDir, mid)
    return alist

alist = [5,2,7,1,9,3,30,11]
print(mergeSort(alist));
```

Atividades

- Implemente o método merge sort para ordenar as listas abaixo:
  - A) [2,5,9,22,1]
  - B) [1,2,5,9,22]
  - C) [22,9,5,2,1]
  - D) [22,9,1,2,5]

- Faça prints para mostrar como as listas foram ordenadas.
- Responda a pergunta:
  1) Porque o merge sort trabalha conceitos de lado direito e lado esquerdo?

Fundamentos da Ordenação

- ***Ordenação com Quick Sort***
  - É o método mais eficaz de ordenação baseado em troca, desenvolvido em 1960.
  - Consiste em se dividir o conjunto original em dois subconjuntos separados por um elemento escolhido arbitrariamente, denominado de pivô.
    - Para cada partição, escolhe-se o primeiro elemento da lista esquerda cuja chave seja maior que a chave do pivô, e escolhe-se o primeiro elemento da lista direita cuja chave seja menor que a chave do pivô. Os dois elementos são trocados.
    - Este processo é repetido enquanto existir registros para serem trocados. Os subconjuntos são ordenados separadamente utilizando de forma recursiva o mesmo procedimento descrito anteriormente. Isto é, para cada subconjunto, escolhe-se arbitrariamente um elemento e, em seguida, ordena-se os dois conjuntos separadamente

Fundamentos da Ordenação

- ***Ordenação com Quick Sort***
  - Exemplo: Escolha do pivô

```
Antes do particionamento: [3, 8, 7, 10, 0, 23, 2, 1, 77, 7]

Depois do particionamento: [1, 0, 2, 3, 8, 23, 7, 10, 77, 7]
```

  - Missão: tente identificar um padrão que justifique o particionamento acima.

Exemplo disponível na postagem:
https://joaoarthurbm.github.io/eda/posts/quick-sort/

Fundamentos da Ordenação

- ***Ordenação com Quick Sort***

Para o array $values = [3, 8, 7, 10, 0, 23, 2, 1, 77, 7]$, temos que $pivot = 3$. Vamos iterar no array identificando os elementos menores ou iguais a ele. O primeiro identificado é o valor 0.

values = [3, 8, 7, 10, 0, 23, 2, 1, 77, 7]

Fundamentos da Ordenação

- ***Ordenação com Quick Sort***

Nosso trabalho agora é colocar o valor 0 à frente do pivot. Então, trocamos esse valor com o valor 8 (imediatamente à frente de 3). Note, no estado parcial, que 0 ficou à frente de 3 e 8 assumiu o índice de 0.

values = [3, 0, 7, 10, 8, 23, 2, 1, 77, 7]

Acabou? Não. O próximo elemento menor ou igual ao pivot (3) é 2.

values = [3, 0, 7, 10, 8, 23, 2, 1, 77, 7]

Fundamentos da Ordenação

- ***Ordenação com Quick Sort***

Temos que trazer 2 para a frente de 3. Vamos fazer isso trocando este valor com o valor 7. Veja, no estado parcial, que agora os valores 0 e 2 estão à frente de 3.

values = [3, 0, 2, 10, 8, 23, 7, 1, 77, 7]

Acabou? Não. O próximo elemento menor ou igual ao pivot (3) é 1.

values = [3, 0, 2, 10, 8, 23, 7, 1, 77, 7]

Fundamentos da Ordenação

- ***Ordenação com Quick Sort***

Temos que trazer 1 para a frente de 3. Vamos fazer isso trocando este valor com o valor 10. Veja, no estado parcial, que agora os valores 0, 2 e 1 estão à frente de 3.

values = [3, 0, 2, 1, 8, 23, 7, 10, 77, 7]

Fundamentos da Ordenação

- ***Ordenação com Quick Sort***

E agora? Agora não há mais elementos menores ou iguais ao pivot para serem identificados. Todos os elementos menores ou iguais (0, 2 e 1) estão imediatamente à frente dele. Então, basta trocarmos o pivot (3) com o último deles (1).

values = [1, 0, 2, 3, 8, 23, 7, 10, 77, 7]

Feito. Agora 3 está em seu lugar, com todos os elementos menores ou iguais à sua esquerda e os elementos maiores à direita.

Atividades

- Considere a estrutura de dados do Quick Sort abaixo. Interprete-o e descreva textualmente seu funcionamento.
  - Prox slide...

Atividades

```
def quickSort(alist):
    N = len(alist)
    def recursiveQuickSort(alist,posMin,posMax):
        if(posMin<posMax):
            p=partition(alist,posMin,posMax)
            recursiveQuickSort(alist,posMin,p-1)
            recursiveQuickSort(alist,p+1,posMax)
        return alist
    def partition(alist,posMin,posMax):
        pivot=alist[posMin]
        i=posMin+1
        j=posMax
        while True:
            while(i<posMax and alist[i]<=pivot):
                i=i+1
            while(j>posMin and alist[j]>=pivot):
                j=j-1
            if(i<j): swap(alist,i,j)
            if(i>=j): break
        swap(alist,posMin,j)
        return j
```

Atividades

- Implemente a ordenação com quick sort para ordenar as listas abaixo:
  A) [2,5,9,22,1]
  B) [1,2,5,9,22]
  C) [22,9,5,2,1]
  D) [22,9,1,2,5]

- Faça prints para mostrar como as listas foram particionadas e, na sequência, ordenadas.
- Responda a pergunta:
  1) Qual a principal regra a ser respeitada pelo particionamento?

Fundamentos da Ordenação

- Encerramento. Quick sort com particionamento e ordenação divididos.

```
def partition (alist, posMin, posMax):
        pivot = alist[posMin]
        i = posMin + 1
        j =  posMax
        while True:
            while(i < posMax and alist[i] <= pivot):
                i = i+1
            while(j>posMin and alist[j] >= pivot):
                j = j-1
            if(i<j):
                swap(alist,i,j)
            if(i>=j):
                break
        swap(alist, posMin, j)
        return j

def recursiveQuickSort(alist, posMin, posMax):
        if(posMin<posMax):
            p=partition(alist, posMin, posMax)
            recursiveQuickSort(alist, posMin, p-1)
            recursiveQuickSort(alist, p+1, posMax)
        return alist
```

Fundamentos da Ordenação

- Encerramento. Quick sort com particionamento e ordenação divididos.

```
alist = [5,2,7,1,9,3,30,11]
#Imprime lista desordenada
print("Desordenada", alist)
# Chama particiona
partition(alist, 0, 7);
# Imprime lista particionada
print("Particionada", alist);
# Chama ordenação
recursiveQuickSort(alist, 0, 7);
print("Ordenada Final", alist);
```

Fundamentos da Ordenação

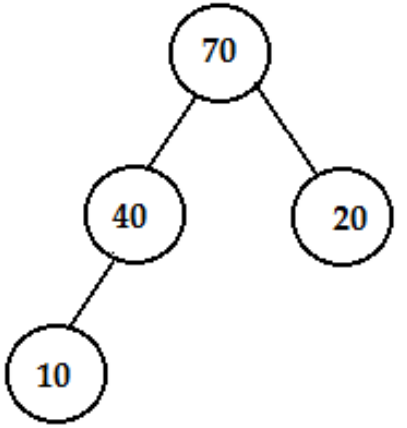
- Encerramento. Quick sort TAD.

```
#Quick sort
def quickSort(alist):
    N = len(alist)
    recursiveQuickSort(alist, 0, N-1);

def partition (alist, posMin, posMax):
    pivot = alist[posMin]
    i = posMin + 1
    j =  posMax
    while True:
        while(i < posMax and alist[i] <= pivot):
            i = i+1
        while(j>posMin and alist[j] >= pivot):
            j = j-1
        if(i<j):
            swap(alist,i,j)
        if(i>=j):
            break
    swap(alist, posMin, j)
    return j

def recursiveQuickSort(alist, posMin, posMax):
    if(posMin<posMax):
        p=partition(alist, posMin, posMax)
        recursiveQuickSort(alist, posMin, p-1)
        recursiveQuickSort(alist, p+1, posMax)
    return alist
```

```
# Chama algoritmo
quickSort(alist);
print("Ordenada Final", alist);
```

Fundamentos da Ordenação

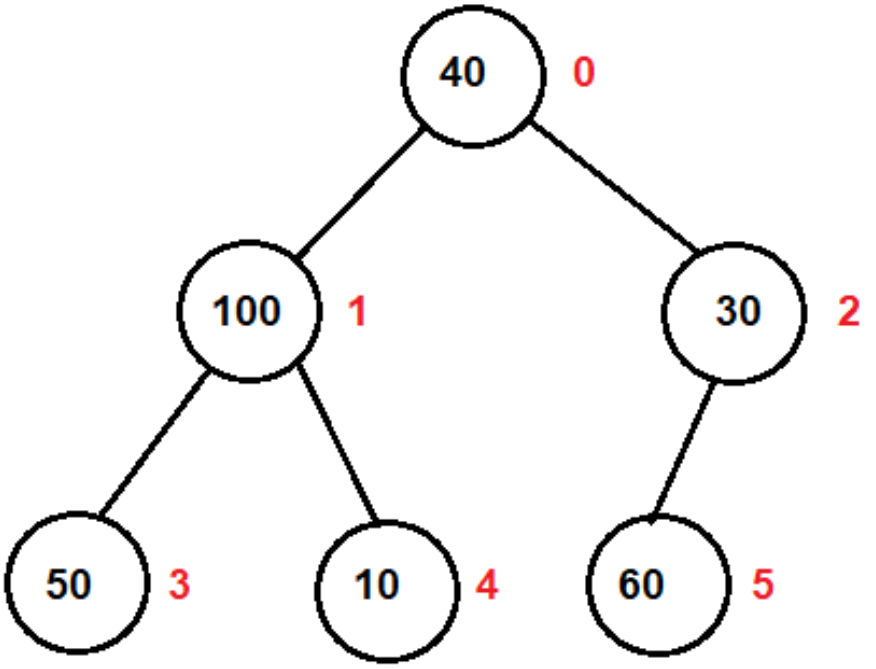
- Heap Sort
  - O heap é uma estrutura de dados definida como uma sequência de itens com chaves c[1], c[2], ..., c[n] tal que c[i] > = c[2i] e c[i] >= c[2i + 1], ou seja, os pais são sempre maiores que os filhos.
  - Um heap descendente (max heap ou árvore descendente parcialmente ordenada) de tamanho n como uma árvore binária de n nós de tal que o conteúdo de cada nó seja menor ou igual ao conteúdo de seu pai.



70	40	20	10
1	2	3	4

Fundamentos da Ordenação

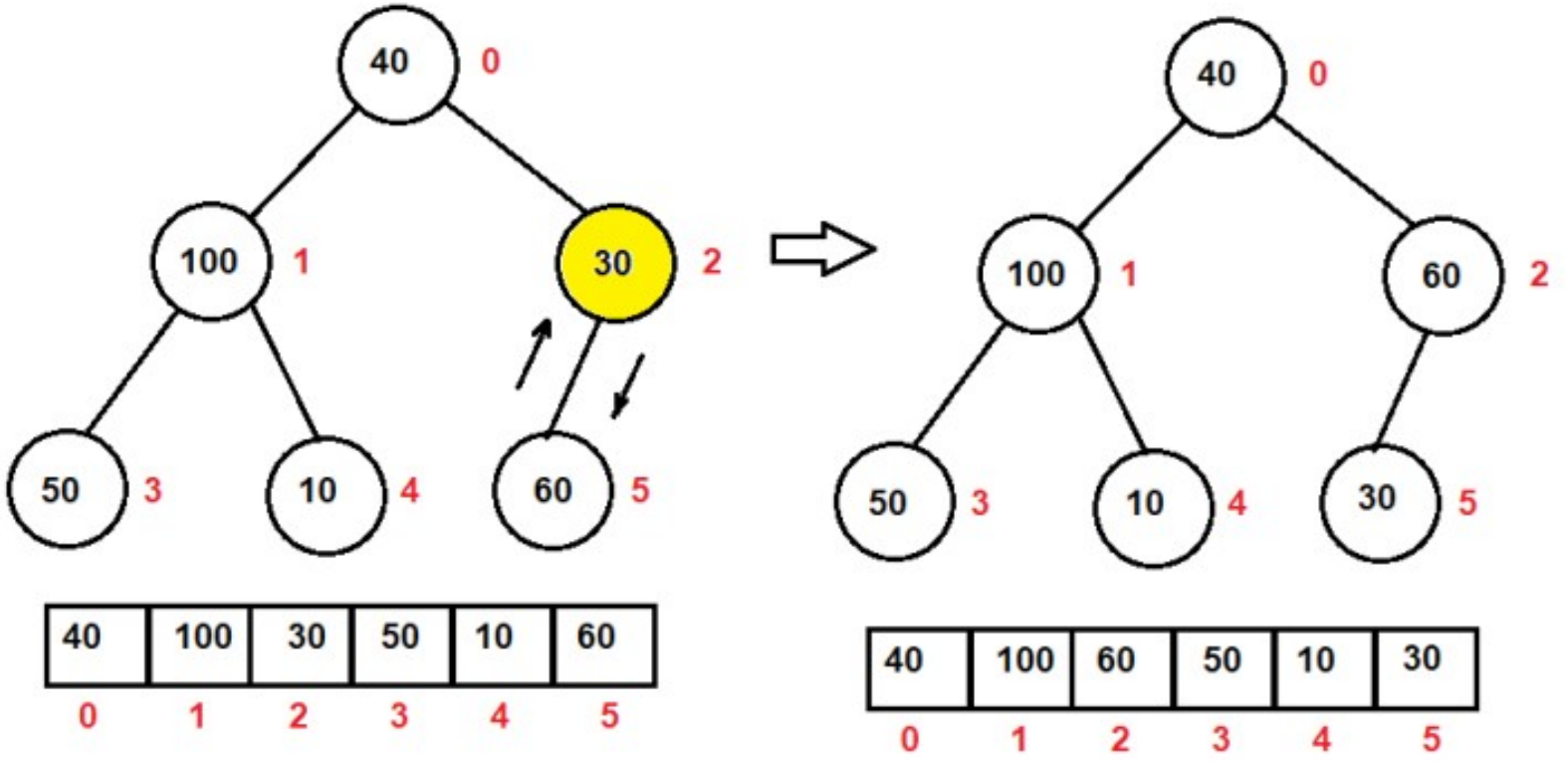
- Heap Sort
  - Heapfy. Considere a lista abaixo:



Fundamentos da Ordenação

- Heap Sort - heapify

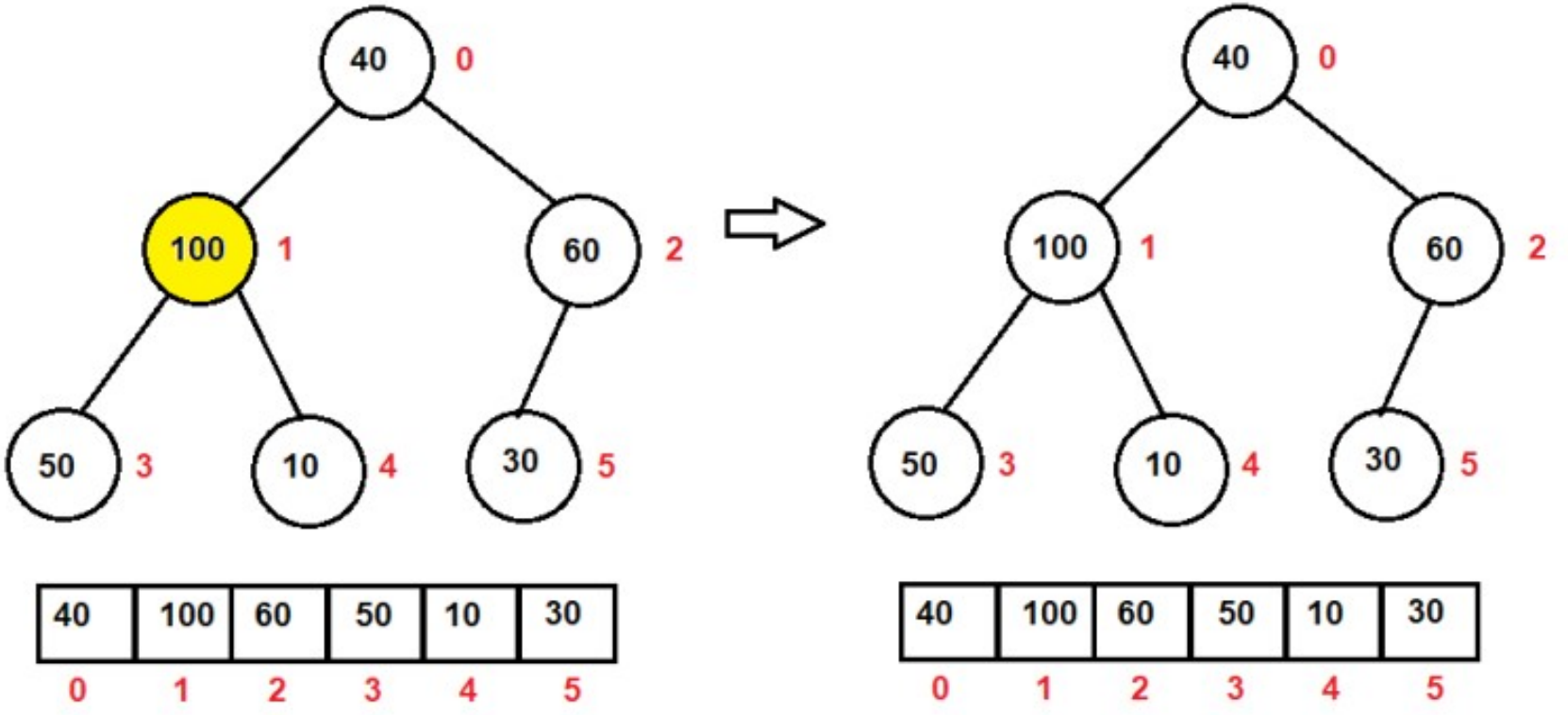
Aplicando o método heapify para o índice 2 (6 // 2 - 1)



Fundamentos da Ordenação

- Heap Sort - heapify

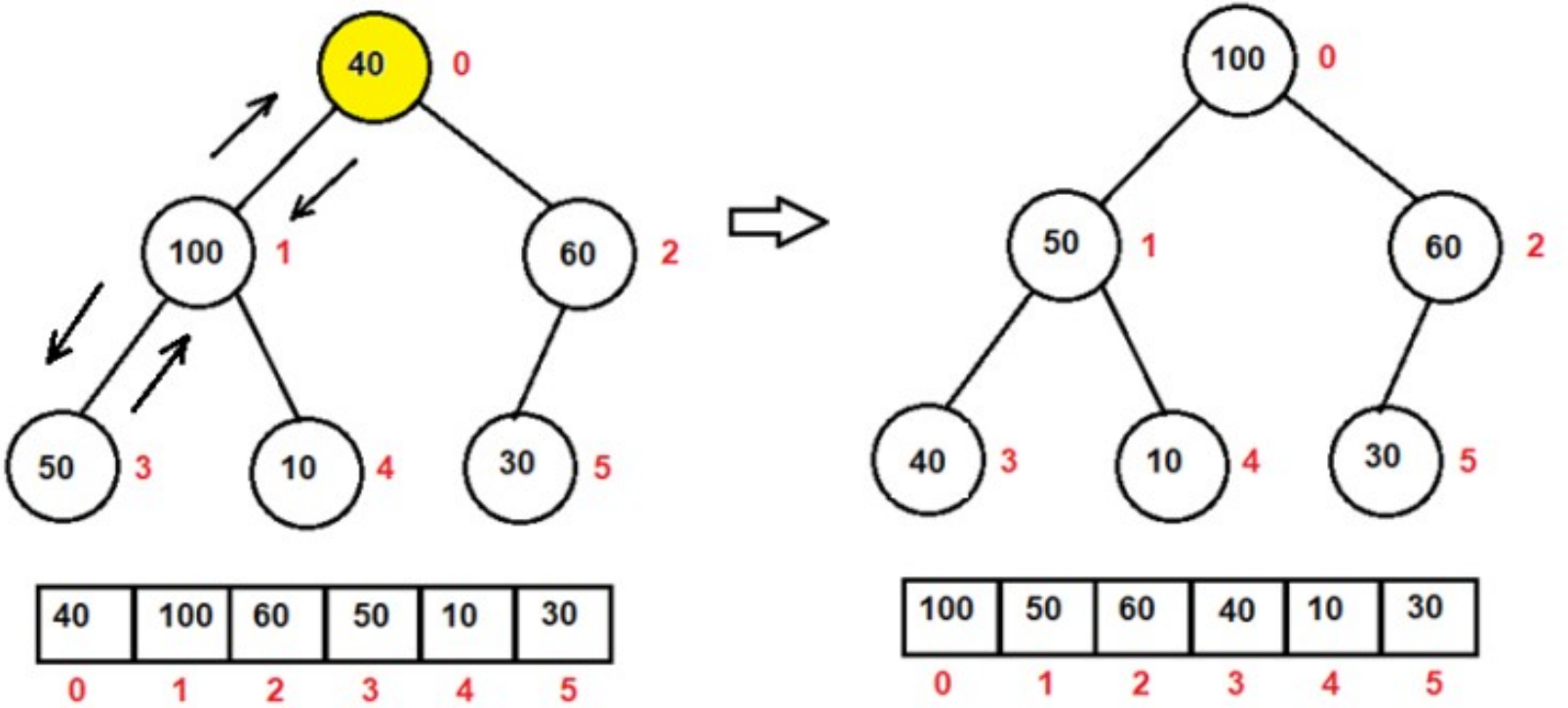
Aplicando o método heapify para o índice 1:



Fundamentos da Ordenação

- Heap Sort - heapify

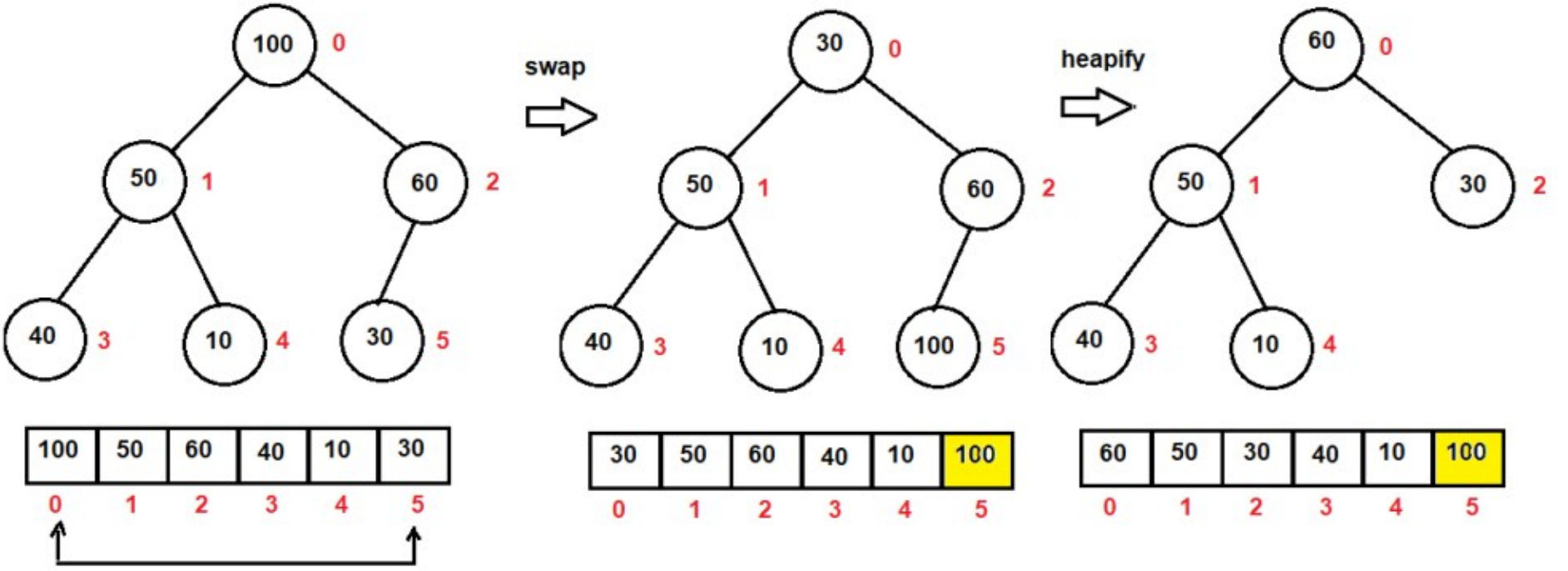
Aplicando o método heapify para o índice 0:



Fundamentos da Ordenação

- Heap Sort

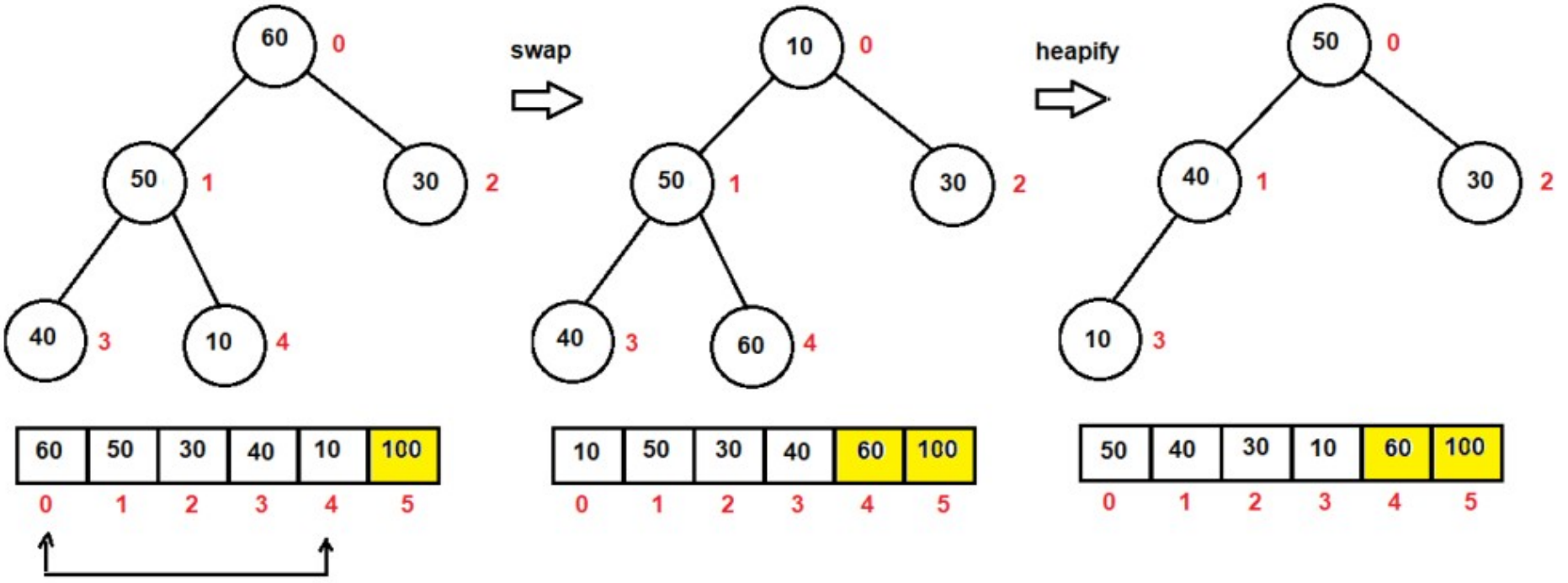
Para n = 5, trocar a[5] com a[0] e aplicar o procedimento heapify para n = 5



Fundamentos da Ordenação

- Heap Sort

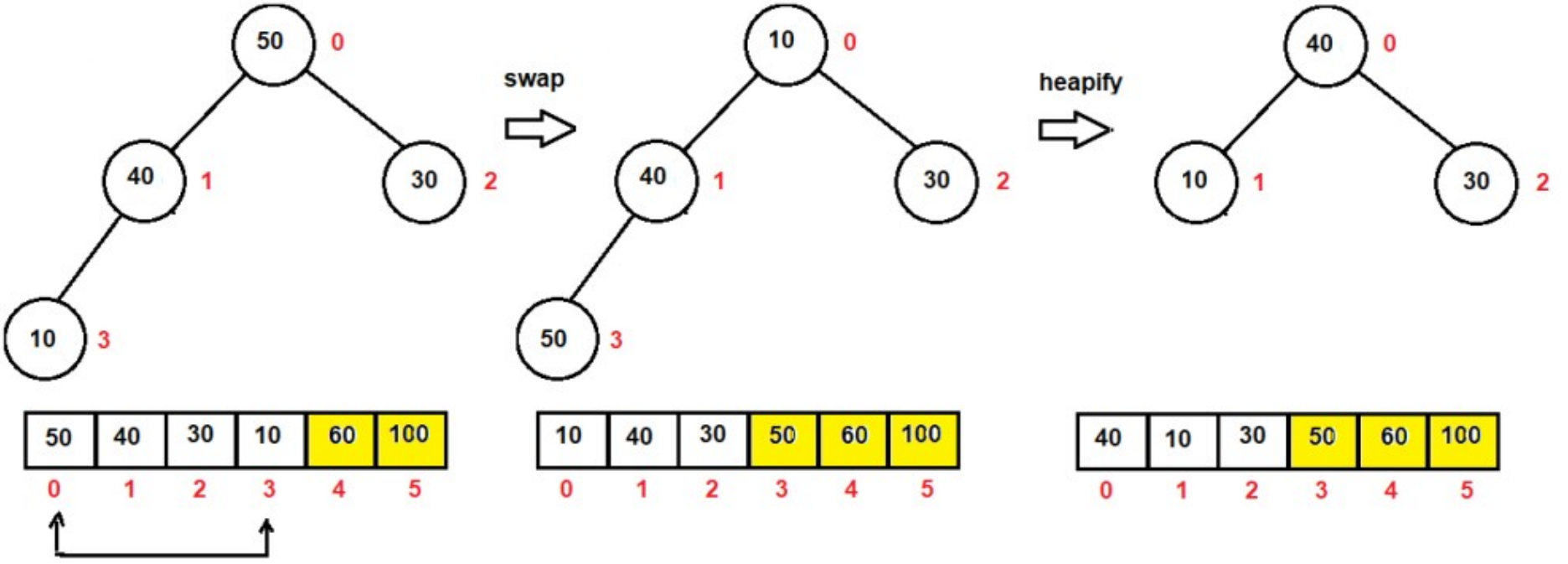
```
Para n = 4, trocar a[4] com a[0] e aplicar o procedimento
heapfiy para n = 4
```



Fundamentos da Ordenação

- Heap Sort

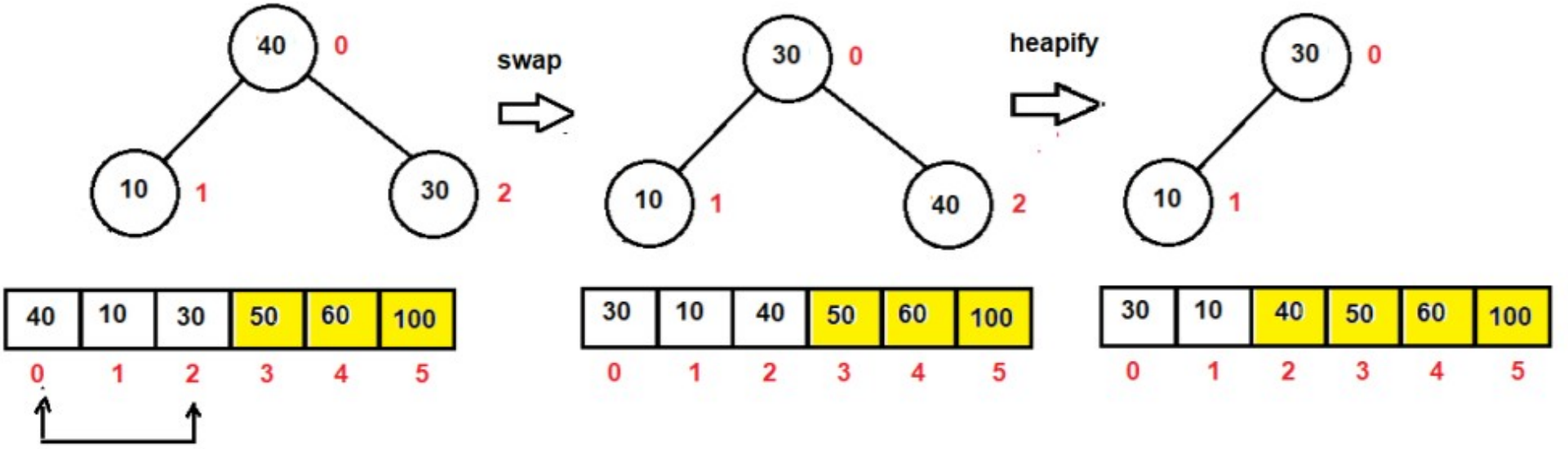
Para n = 3, trocar a[3] com a[0] e aplicar o procedimento heapify para n = 3



Fundamentos da Ordenação

- Heap Sort

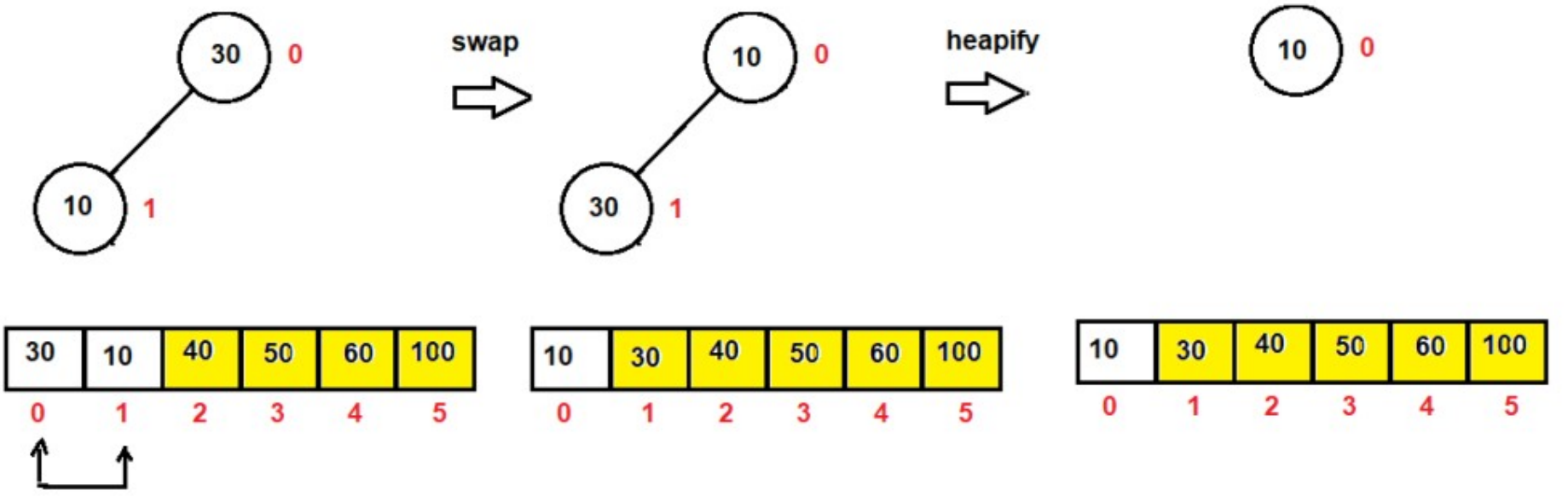
```
Para n = 2, trocar a[2] com a[0] e aplicar o procedimento
heapify para n = 2
```



Fundamentos da Ordenação

- Heap Sort

```
Para n = 1, trocar a[1] com a[0] e aplicar o procedimento
heapify para n = 1
```



Fundamentos da Ordenação

- Heap Sort

```
# Heapify
def heapify(arr, n, i):
        largest = i # Inicializa maior como root
        l = 2 * i + 1 # left
        r = 2 * i + 2 # right
# Veja se o filho esquerdo do root existe e é
# maior que a raiz
        if l < n and arr[i] < arr[l]:
                largest = l
# Veja se o filho certo da raiz existe e é
# maior que a raiz
        if r < n and arr[largest] < arr[r]:
                largest = r
# Altere o root, se necessário
        if largest != i:
                (arr[i], arr[largest]) = (arr[largest], arr[i]) # swap
# Heapify o root.
                heapify(arr, n, largest)
```

Fundamentos da Ordenação

- Heap Sort

```
# Heap
def heapSort(arr):
        n = len(arr)
# Construa o maxheap.
# Como o último pai estará em ((n//2)-1), podemos começar nesse local.
        for i in range(n // 2 - 1, -1, -1):
                heapify(arr, n, i)
# Um por um extrai elementos
        for i in range(n - 1, 0, -1):
                (arr[i], arr[0]) = (arr[0], arr[i]) # swap
                heapify(arr, i, 0)
```

```
# Execução
arr = [12, 11, 13, 5, 6, 7, ]
heapSort(arr)
print('Ordenado')
print (arr)
```

Atividades

- Implemente a ordenação com heap sort para ordenar as listas abaixo:
  A) [2,5,9,22,1]
  B) [1,2,5,9,22]
  C) [22,9,5,2,1]
  D) [22,9,1,2,5]

- Faça prints para mostrar como as listas foram particionadas e, na sequência, ordenadas.
- Responda a pergunta:
  1) Qual a importância do heapify?