Capítulo

5 Ordenação

Ordenação Interna

Ordenar é o ato de rearranjar um certo conjunto de objetos em uma ordem ascendente ou descendente. O ato de ordenar é um ato muito utilizado no nosso cotidiano. 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 a função ***busca*** ilustrada no Fragmento de Código 5.1. Esta função retorna o endereço da célula cuja chave é igual à chave de pesquisa ou "-1", caso não exista chave igual à chave de pesquisa.

```
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
```

Fragmento de Código 5.1. Implementação em PYTHON da função busca em um vetor de inteiros não ordenado.

Considerando agora o vetor **a**, um conjunto de chaves organizadas sequencialmente, a função ***busca***, ilustrada no Fragmento de Código 5.2, possui uma linha a mais para otimizar esta pesquisa, pois, se o conjunto é ordenado e a chave de pesquisa é menor que a chave do endereço corrente, logo, não se faz necessário continuar a pesquisa, pois a chave certamente não existe no vetor, uma vez que o vetor está ordenado.

```
# 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
```

Fragmento de Código 5.2. Implementação em PYTHON da função busca em um vetor de inteiros ordenado.

O Python tem uma característica, muito importante que é a de Poliformismo, isto é se o objeto for inteiro ele compara na forma de um inteiro, se o objeto for um string ele compara como um string. Ou seja, se mudarmos o tipo de entrada, o procedimento funcionará da mesma forma. É importante frisar que se o objeto for de uma classe não primitiva, deve-se implementar os métodos de comparação na classe.

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. Estes métodos são métodos baseados em troca descritos a seguir.

Ordenação por Troca

Os métodos de ordenação baseados em 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***.

A visão esquemática do procedimento troca é mostrada na figura 2.2. O procedimento troca tem como objetivo trocar os elementos que estão nas posições, respectivamente, "i" e "j" .

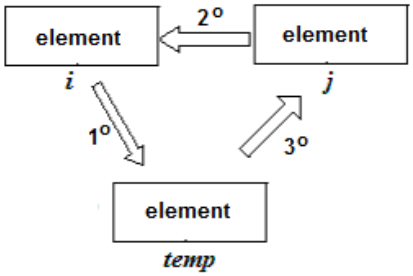


Figura 5.1 – Visão esquemática do procedimento troca.

O fragmento de código 5.5 ilustra uma forma de se implementar o procedimento *troca (swap).*

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

Fragmento de Código 5.5. Implementação em PYTHON do procedimento troca (swap).

O fragmento de código 5.5 ilustra uma outra forma de se implementar o procedimento *troca (swap).*

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

Fragmento de Código 5.5. Outra forma de Implementação em PYTHON do procedimento troca (swap).

Ordenação por Bolha (BubbleSort)

Este método de ordenação por troca é o mais simples, embora não muito eficaz. 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 c_i e c_{i+1} são respectivamente as chaves dos registros corrente e o próximo.

Ordenar o conjunto de chaves 30, 10, 50, 40 , 22 utilizando o método de seleção *BubbleSort*. As chaves em negrito correspondem a sequência de execução.

Observe na Figura 5.2 que na 1ª passada a "bolha" vai da 1ª célula até a célula de endereço n -1, onde n é o número de células, pois a chave da célula n, já possui a maior chave. Na 2ª passada a "bolha" vai da 1ª célula até a célula n -2, pois as duas últimas chaves já estão no seu lugar. Pode-se observar também que na 3ª passada a "bolha" vai da 1ª célula até a célula n -3, pois as três últimas chaves já estão no seu lugar. Finalmente, na 4ª passada a "bolha" vai da 1ª célula até a célula n -4, pois as três últimas chaves já estão no seu lugar.

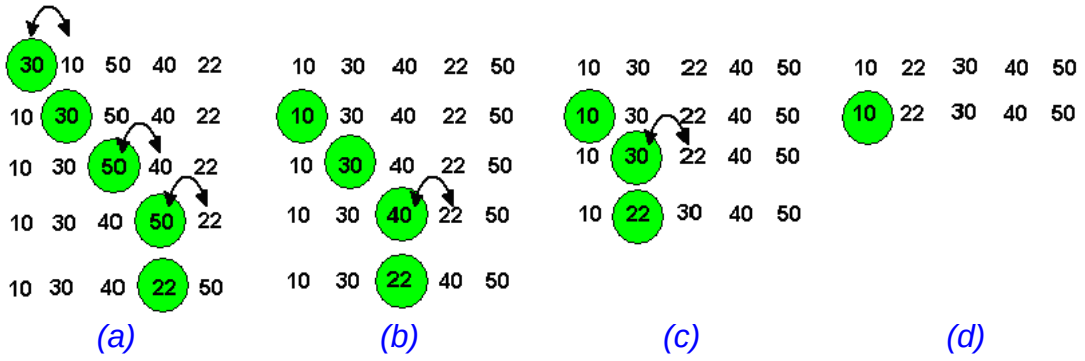


Figura 5.2 – Visão esquemática do procedimento bubbleSort. A letra a) corresponde à 1ª passada, a letra b) corresponde à 2ª passada, a letra c) corresponde à 3ª passada e, finalmente, a letra d) corresponde à 4ª passada.

O fragmento de código 5.6 ilustra o procedimento traditionalBubbleSort.

```
def traditionBubbleSort(alist):
  N = len(alist)
  for passnum in range(1, N):
      for i in range(0,N-passnum):
          if alist[i]>alist[i+1]:
              sort.swap(alist,i,i+1)
  return alist
```

Fragmento de Código 5.6. Implementação em PYTHON do procedimento raditionalB*ubbleSort*.

É importante também frisar que o programa pararia nesta passada mesmo, pois o algoritmo não efetuaria mais trocas, implicando que o vetor já está ordenado. É fácil observar que o número máximo de passadas é igual a n -1, onde n é o número total de células do vetor, pois, a cada passada coloca-se o maior elemento na posição correspondente. Na primeira passada é colocado o maior elemento do vetor na última posição, na segunda passada coloca-se o segundo maior elemento na penúltima posição, logo, na passada n - 1, os, n – 1 elementos já estão em sua posição correspondente implicando que o primeiro estará também.

IMPORTANTE: O processo termina quando não existir mais possibilidades de trocas, ou quando o número de passadas for igual a n – 1 onde n é o número de células.

Mas, e se em uma determinada passada o vetor já estive ordenado? É fácil concluir que o procedimento não necessitaria completar o número máximo de passadas. Nesse caso, criamos uma variável lógica que indicará se houve ou não trocas em uma determinada passa genérica. Se houve troca o procedimento continua e, no caso contrário o procedimento acaba, ou seja, o vetor já está ordenado.

O fragmento de Código 5.6 ilustra o procedimento *bubbleSort com a opção do corte sinalizando que não houve mais trocas.*

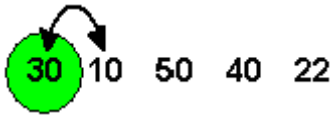
```
def bubbleSort(alist):
  N = len(alist)
  for passnum in range(1,N):
      trocou=False
      for i in range(0,N-passnum):
          if alist[i]>alist[i+1]:
              sort.swap(alist,i,i+1)
              trocou=True
      if(not trocou):break
  return alist
```

Fragmento de Código 5.6. Implementação em PYTHON do procedimento *bubbleSort*.

Ordenação por Seleção(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. O processo é repetido trocando-se o menor valor da chave dos n -1 registros (desconta-se o primeiro) com o segundo registro, troca-se o menor valor da chave dos n - 2 registros (desconta-se o primeiro e o segundo registros) com o terceiro registro e assim sucessivamente.

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

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

O próximo registro é o terceiro, chave 50, e o menor valor a partir dele é o último registro, chave 30. Troca-se o registro de chave igual a 30 com o terceiro elemento de chave igual a 50.

A nova sequência formada será :

10 22 30 40 50

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.

Como o registro corrente é o penúltimo pode-se parar a rotina pois o vetor estará ordenado.

IMPORTANTE: O processo termina quando o registro corrente for igual a n – 1, onde n é o número de células.

O fragmento de Código 5.7 ilustra o procedimento *SelectSort*.

```
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
```

Fragmento de Código 5.7 Implementação em PYTHON do procedimento *SelectSort*

Quick Sort

É um método mais eficaz de ordenação baseado em troca. Este algoritmo foi desenvolvido por C. A. R. Hoare em 1960. O conceito básico deste método 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.

Inicialmente vamos ilustrar o *quicksort* com um exemplo. Considere o vetor: 30, 50, 15 , 60, 35, 70, 20, 90, 25, 70. O primeiro passo é escolher o pivô, que no nosso elemento será sempre o primeiro elemento do vetor. No nosso exemplo o pivô é o elemento de chave igual a 30. Devemos colocar este elemento no lugar onde, todos os elementos cujas chave são menores que a chave do pivô, estejam à esquerda do pivô, e os elementos cujas chave são maiores que a chave do pivô, estejam à direita do pivô. O vetor resultante é o vetor: 20, 25, 15, 30, 35, 70, 60, 90, 50, 70. A figura 5.8, na segunda linha, ilustra este procedimento. A seguir o vetor é particionado em dois subconjuntos, e o algoritmo é executado novamente nestes dois vetores. Este procedimento é executado recursivamente, até que os subconjuntos tenham um só elemento.

O objetivo da partição é permitir que um elemento específico encontre sua posição correta no vetor. Uma maneira eficiente de implementar a partição é a seguinte. Considere o pivô o elemento v[min], onde min é o endereço da chave inicial do vetor considerado. Dois ponteiros i e j são inicializados com os respectivos valores do limite mínimo e máximo do vetor. Os dois ponteiros são movidos um em direção ao outro de acordo com os seguintes passos.

Passo 1 : Incrementar o ponteiro i em uma posição até que v[i].chave >= pivô
Passo 2 : Decrementar o ponteiro j em uma posição até que v[j].chave <= pivô
Passo 3 : Se i < j, troque v[i] com v[j]

O processo é repetido até que a condição descrita no passo 3 falhe. Neste ponto v[j] será trocado com v[min].

A seguir vamos ilustrar este procedimento com exemplo.

pivô = v[min].chave = 25

```
i ➔                          j
25  57  48  37  12  92  86  33
    i                        j
25  57  48  37  12  92  86  33
    i                    ← j
25  57  48  37  12  92  86  33
    i                ← j
25  57  48  37  12  92  86  33
    i            ← j
25  57  48  37  12  92  86  33
    i            j
25  57  48  37  12  92  86  33
```

Será feita a troca : v[i] com v[j]

```
     i           j
25  12  48  37  57  92  86  33
```

Continuando o processo :

```
        i ➔      j
25  12  48  37  57  92  86  33
         i       ← j
25  12  48  37  57  92  86  33
         i   ← j
25  12  48  37  57  92  86  33
       ← j
25  12  48  37  57  92  86  33
    j   i
25  12  48  37  57  92  86  33
```

Neste ponto j > i troca v[min] com v[j]

```
12  25  48  37  57  92  86  33
```

O elemento pivô, chave igual a 25, está na posição correta. Repete-se o processo com os subvetores de forma recursiva (12) e (48 37 57 92 86 33). Este algoritmo está implementado no Fragmento de Código 5.8.

```
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
```

Fragmento de Código 5.8. Implementação em PYTHON do procedimento *quicksort*.

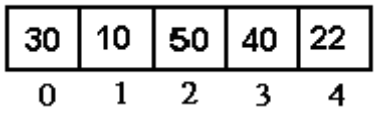
Ordenação por Inserção

O conceito básico deste método consiste em inserir um registro ***R*** numa sequência de registros ordenados R_1, R_2, R_3, ... R_n, de chaves $C_1 \leq C_2 \leq C_3$, .. $\leq C_n$, sendo que a nova sequência de **n** + 1 termos deverá ficar também ordenada.

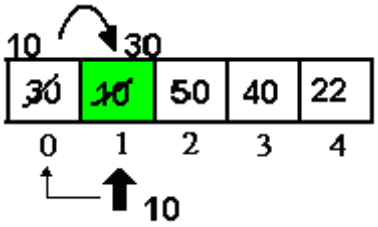
A ideia básica está em varrer o vetor do segundo elemento até o fim e, para cada elemento, inseri-lo na lista formada com os elementos anteriores de tal forma que esta lista continue ordenada A seguir descreve-se um exemplo para ilustrar este conceito.

Exemplo 1: Ordenar o conjunto de chaves 30, 10, 50, 40 , 22 utilizando o método de inserção. A primeira célula corresponde ao menor valor do vetor (-∝). Este valor deverá ser armazenado nesta célula para que o procedimento funcione.

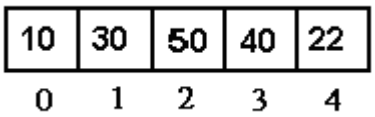
Sequência original



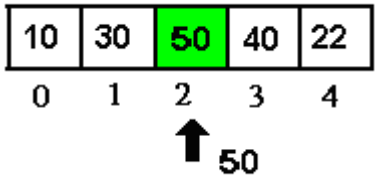
Pode-se observar na figura anterior que a primeira linha corresponde ao conjunto dado. O procedimento básico consiste em caminhar sobre o vetor a partir do segundo elemento, e ordenar os elementos anteriores ao elemento corrente trocando-os de posição. Por exemplo, o 2º elemento é o 10, a lista anterior (-∝ , 30). A ideia é incluir o 10 na lista anterior e continuar ordenada. A figura seguinte é o resultado desta troca.



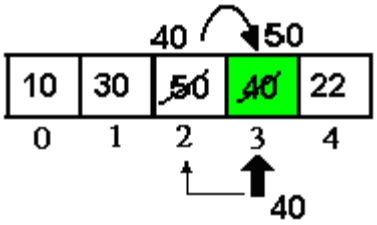
A sequência após as trocas é a seguinte:



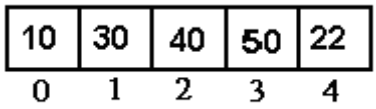
O próximo elemento é o 50, e a lista anterior (-∝ , 10, 30). A ideia é incluir o 50 na lista anterior e continuar ordenada. Pode-se observar que não haverá trocas, pois o 50 é o maior elemento da lista. Logo o vetor permanecerá inalterado



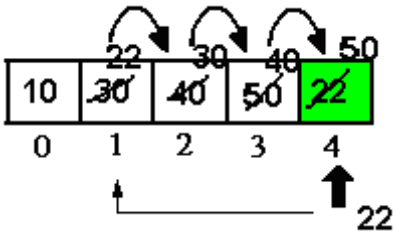
O próximo elemento é o 40, e a lista anterior (-∝ , 10, 30, 50). A ideia é incluir o 40 na lista anterior e continuar ordenada. A figura seguinte é o resultado desta troca.



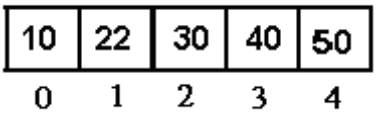
A sequência após as trocas é a seguinte:



O próximo elemento é o 22, e a lista anterior (-∝ , 10, 30, 40, 50). A ideia é incluir o 22 na lista anterior e continuar ordenada. A figura seguinte é o resultado desta troca.



A sequência após as trocas, que será o vetor ordenado, é a seguinte:



O método de inserção sort está implementado no Fragmento de Código 5.9. A seguir descreve-se a definição da estrutura de dados utilizada no procedimento.

Neste procedimento faz-se necessário acessar um elemento da tabela hipotético com índice 0 (zero). Logo, na definição da estrutura de dados o vetor deverá variar de 0 até N (onde o número de elementos será N – 1).

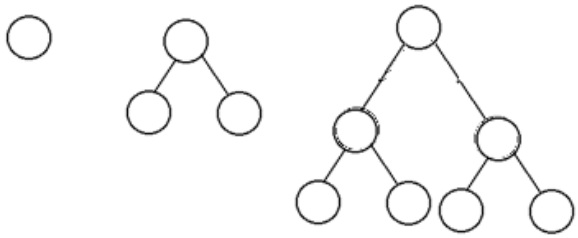
```
def insertSort(alist):
  N = len(alist)
  for i in range(1,N):
    aux=alist[i]
    j=i-1
    # Achar a posição do menor elemento
    while( j >= 0  and alist[j] >= aux):
        alist[j+1]=alist[j]
        j=j-1
    alist[j+1]=aux
  return alist
```

Fragmento de Código 5.9. Implementação em PYTHON da rotina insertSort

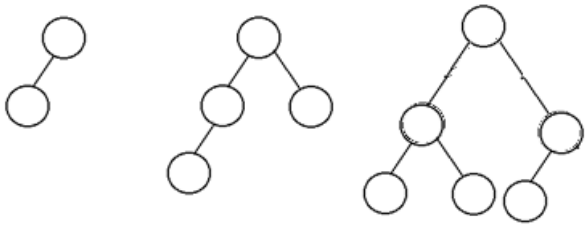
Heaps

O ***heap*** é uma estrutura de dados proposta por Willian (1964) e é definido como uma sequência de itens com chaves c[1], c[2], ..., c[n] tal que $c[i] \geq c[2i]$ e $c[i] \geq c[2i + 1]$ $\forall$ i = 1,2,3,..., n/2.

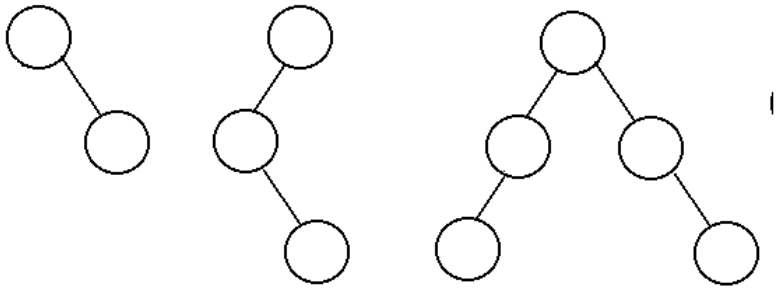
Exemplos de estruturas heaps como árvore completa



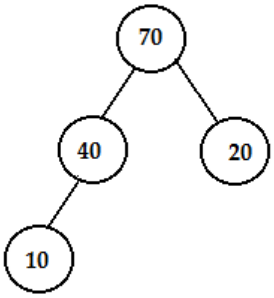
Exemplos de estruturas heaps como árvore incompleta. O nível incompleto tem que ser o último nível da árvore e, as células ajustadas à esquerda.



Exemplos de estruturas que NÃO são heaps



Um **heap descendente** (conhecido também como **max heap** ou **árvore descendente parcialmente ordenada**) de tamanho *n* como uma árvore binária completa (incompleta*), de *n* nós de tal que o conteúdo de cada nó seja menor ou igual ao conteúdo de seu pai.



Pode-se concluir que, devido a definição anterior, um heap descendente, possui como raiz o maior valor da chave do vetor. Também pode-se concluir que qualquer percurso no heap que não inclua mais de um nó em qualquer nível é uma lista linear descendente.

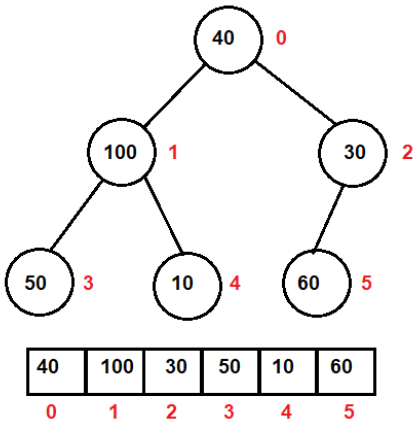
- O ***maxHeap*** é uma estrutura de dados e é definido como uma sequência de itens com chaves c[1], c[2], ..., c[n] tal que:

$$c[i] \geq c[2i] \text{ e } c[i] \geq c[2i+1] \quad \forall\, i = 1,2,3,..., n/2$$

70	40	20	10
1	2	3	4

Construindo um max heap usando o procedimento heapfy

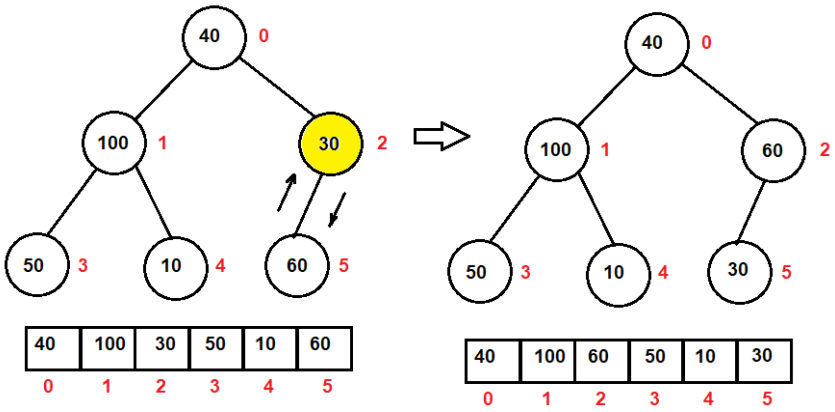
Considere o vetor e a representação desse vetor numa estrutura heap correspondente



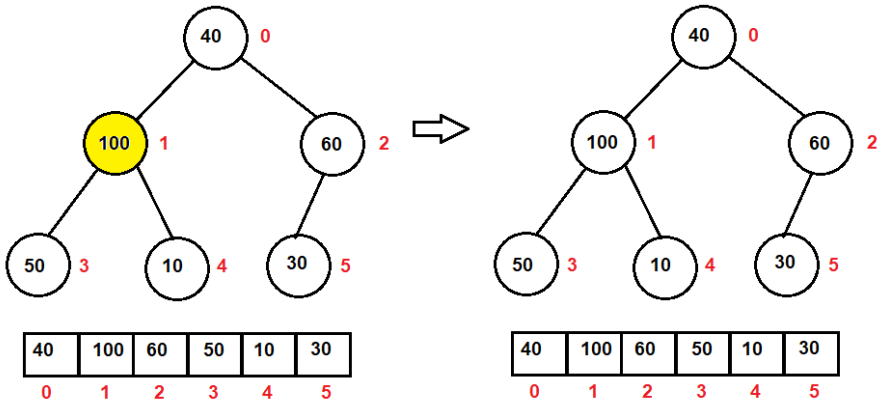
O número fora da célula, em vermelho, representa os índices do array dos dados.

Aplica-se o método heapify do índice inicial igual a (n // 2 - 1), onde n é o tamanho do vetor, até zero.

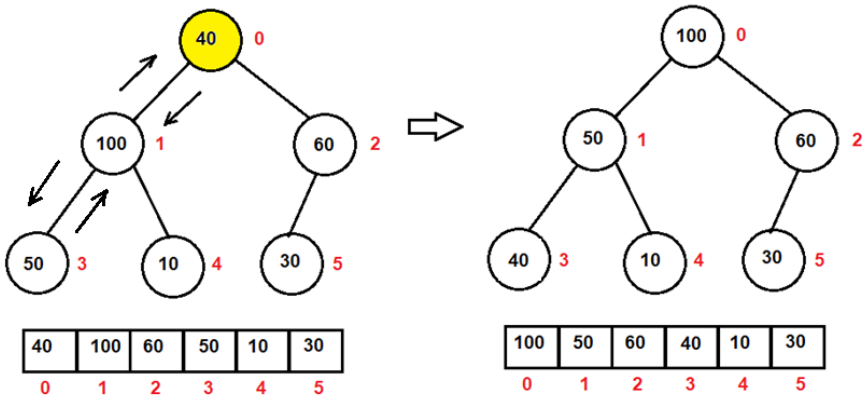
Aplicando o método heapify para o índice 2 (6 // 2 - 1)



Aplicando o método heapify para o índice 1:



Aplicando o método heapify para o índice 0:

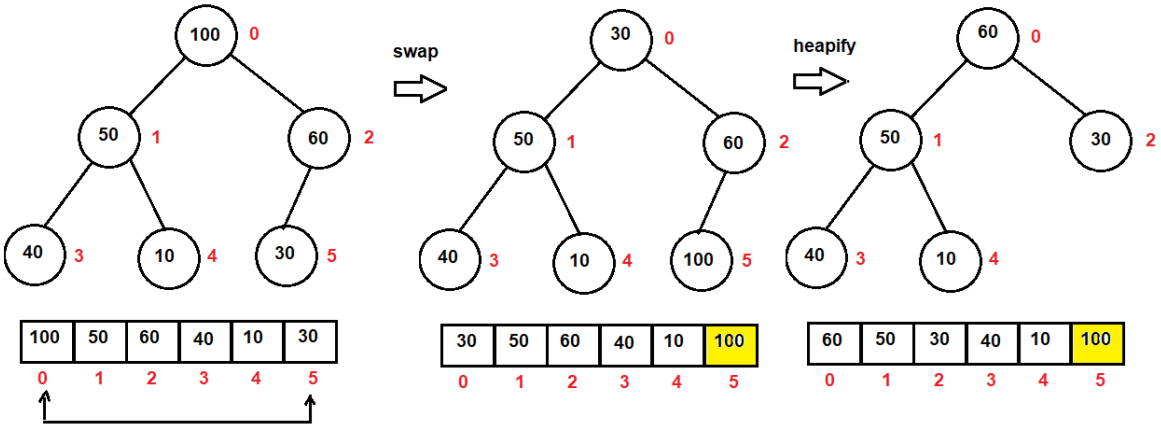



```
# To heapify subtree rooted at index i.
# n is size of heap
def _heapify(arr, n, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[i] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap
        # Heapify the root.
        _heapify(arr, n, largest)
```

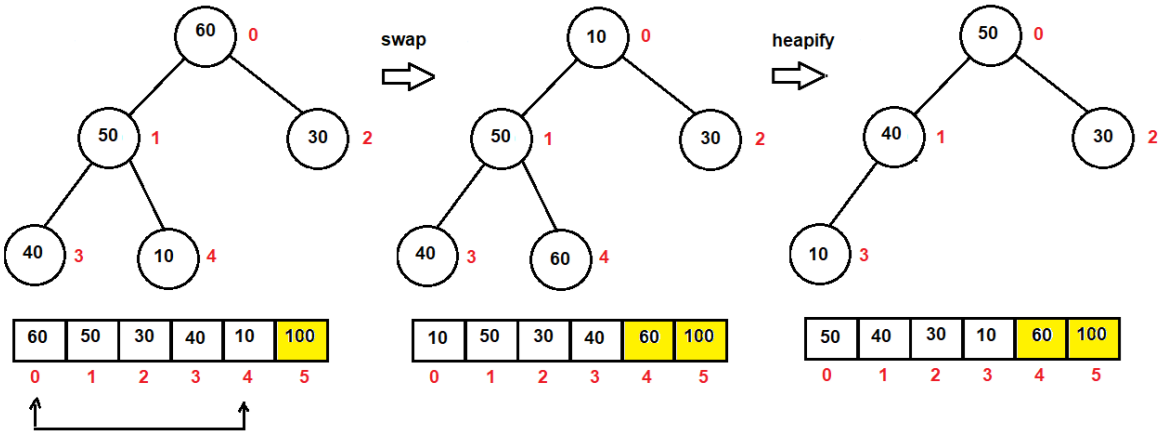
Ordenando

Considerando o heap já criado, trocamos as células de índice 0 e n (n= len(array)- 1) e aplicamos o procedimento heapfy novamente, até que n seja igual a 1.

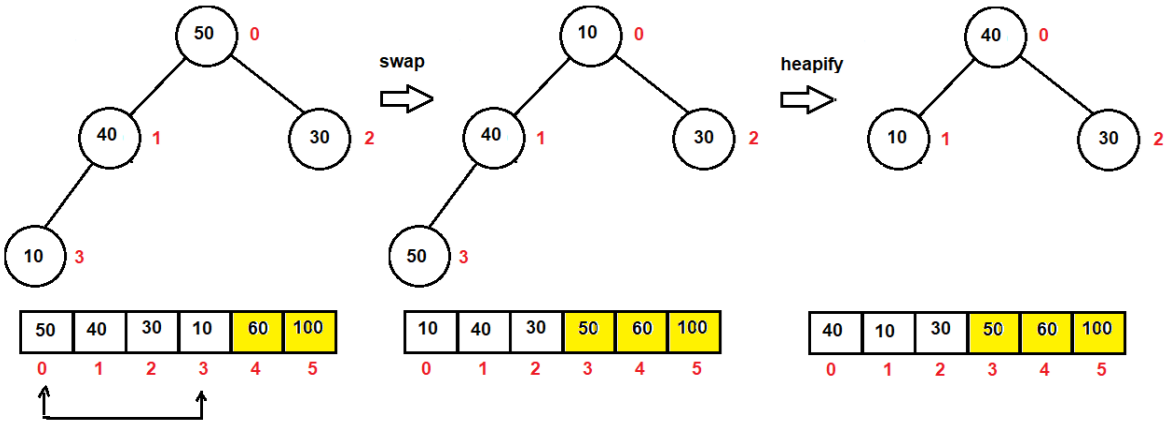
Para n = 5, trocar a[5] com a[0] e aplicar o procedimento heapify para n = 5



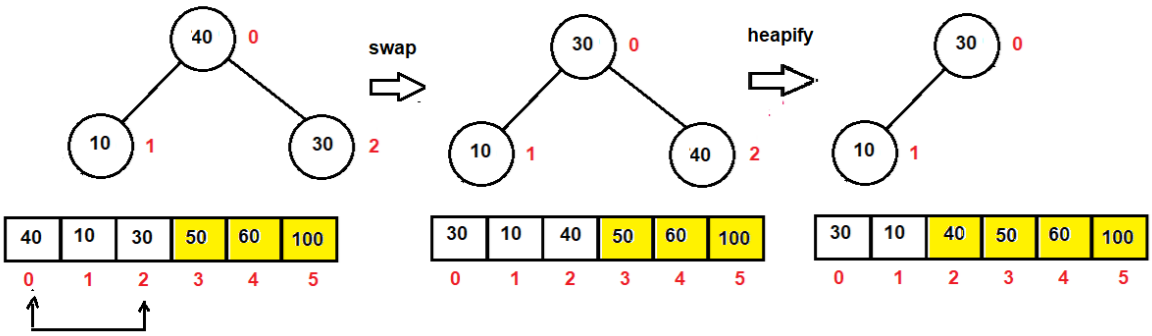
Para n = 4, trocar a[4] com a[0] e aplicar o procedimento heapfiy para n = 4



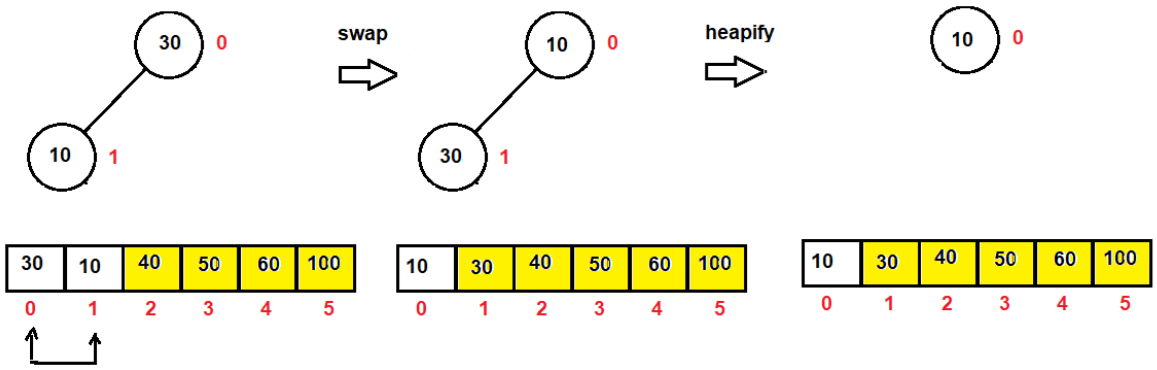
Para n = 3, trocar a[3] com a[0] e aplicar o procedimento heapify para n = 3



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



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

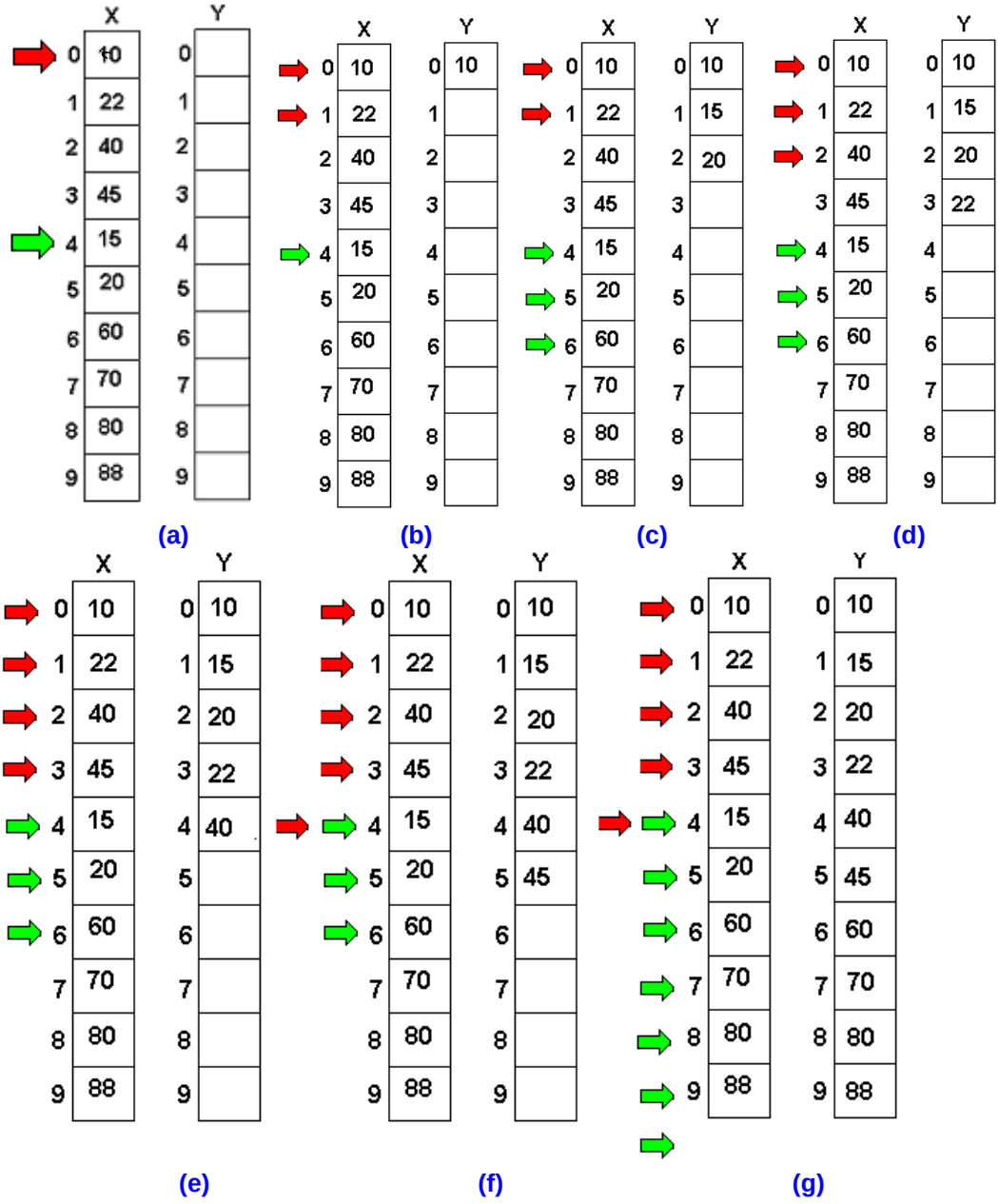



```
# The main function to sort an array of given size
def heapSort(arr):
    # https://www.programiz.com/dsa/heap-sort
    n = len(arr)

    # Build a maxheap.
    # Since last parent will be at ((n//2)-1)
    # we can start at that location.
    for i in range(n // 2 - 1, -1, -1):
        _heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]   # swap
        _heapify(arr, i, 0)
```

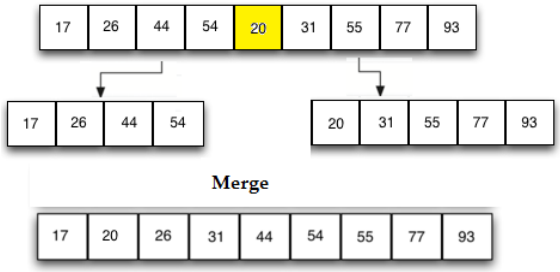
Merge

Este método consiste em intercalar dois vetores ordenados, gerando-se no final um novo vetor também ordenado.

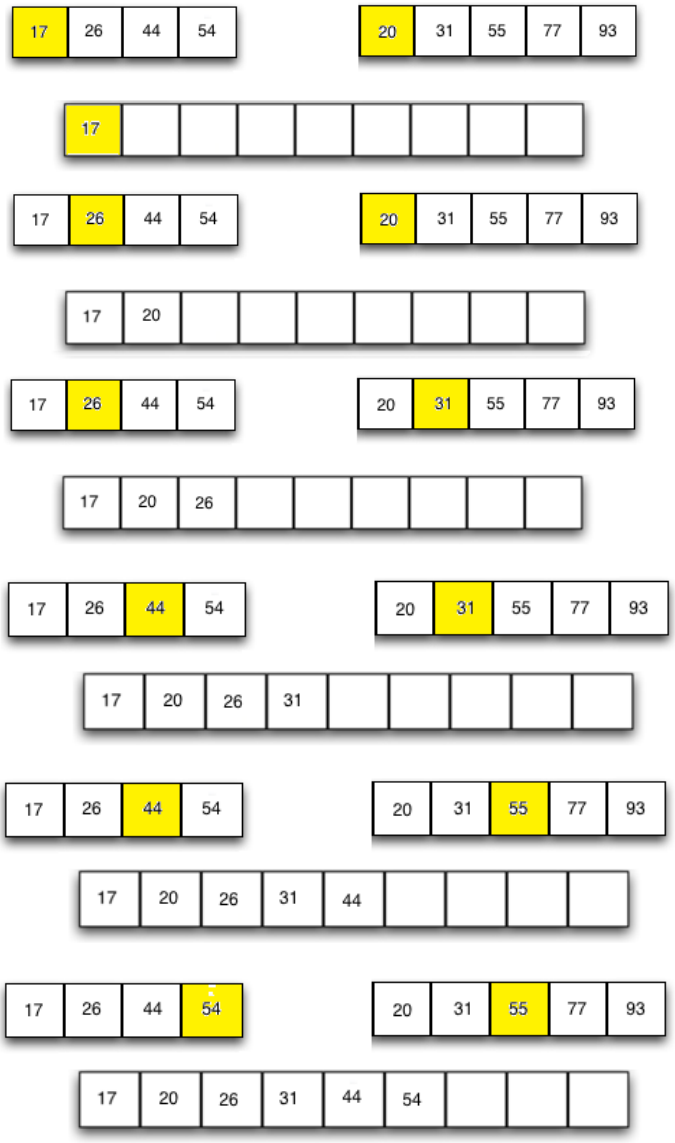
Vamos considerar um vetor X, contendo dois arquivos lógicos (dois vetores). O primeiro arquivo lógico começa na posição 0 e termina na posição 5. O segundo arquivo lógico começa na posição 4 e termina na posição 9. Considere também o arquivo Y que conterá os dois arquivos lógicos.

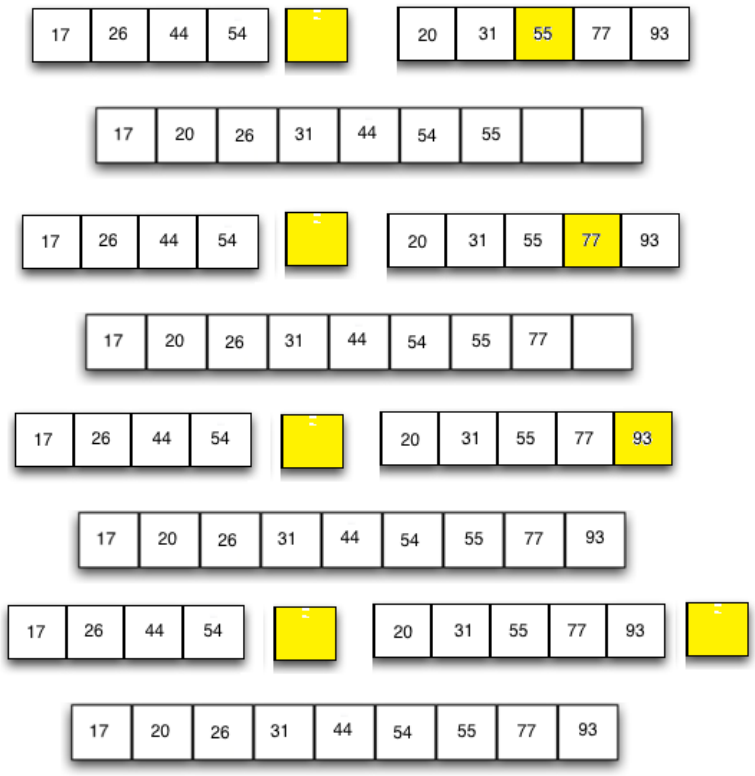


Exemplo de um merge em um vetor. Considerando dois arquivos lógicos ordenados: o primeiro do índice 0 até 3, e, o segundo do índice 4 até o final.



Merge passo à passo





O procedimento merge está implementado no Fragmento de Código 5.11.

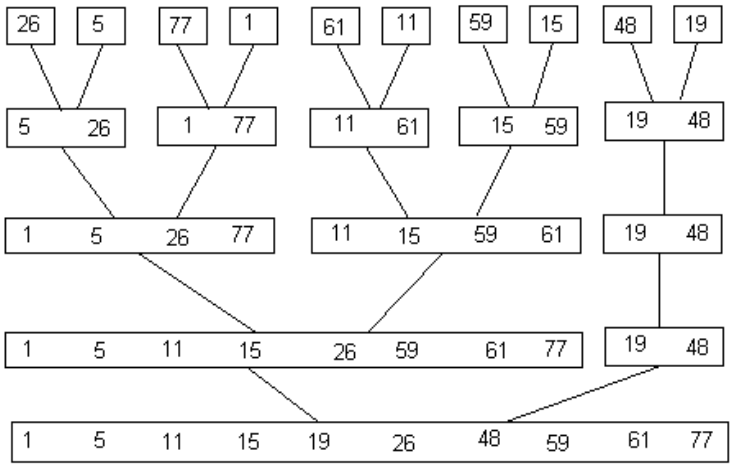
```
def merge(alist,mid):
      lefthalf = alist[:mid]
      righthalf = alist[mid:]
      i=j=k=0
      while i < len(lefthalf) and j < len(righthalf):
          if lefthalf[i] <= righthalf[j]:
              alist[k]=lefthalf[i]
              i += 1
          else:
              alist[k]=righthalf[j]
              j += 1
          k += 1
      while i < len(lefthalf):
          alist[k]=lefthalf[i]
          i += 1
          k += 1

      while j < len(righthalf):
          alist[k]=righthalf[j]
          j += 1
          k +  1
      return(alist)
```

Fragmento de Código 5.11. Implementação em PYTHON do método merge.

Merge Sort

O conceito básico deste método consiste em sortear n listas de tamanho 1. Estas listas são intercaladas, utilizando-se o procedimento Merge, em n/2 pares de listas, cada qual com tamanho 2 (se n é ímpar, então existirá uma lista com tamanho 1). Estas n/2 listas são intercaladas por pares gerando-se no final, como resultado, uma lista. De forma recursiva a lista é dividida em n/4 listas são intercaladas por pares gerando-se no final, como resultado, uma nova lista e assim por diante até se ter uma lista somente que está ordenada.

Ordenar o conjunto de chaves (26, 5, 77, 1, 61, 11, 59, 15, 48, 19) utilizando o método de MergeSort.



O procedimento interativeMergeSort não recursivo, está implementado no Fragmento de Código 5.12.

```
def iterativeMergeSort(alist):
    # curr_size;  For current size of subarrays to be merged
    #             curr_size varies from 1 to n/2
    # left_start; For picking starting index of left subarray
    #             to be merged
    # Merge subarrays in bottom up manner.  First merge subarrays of
    # size 1 to create sorted subarrays of size 2, then merge subarrays
    # of size 2 to create sorted subarrays of size 4, and so on.
    n = len(alist)
    curr_size = 1
    while curr_size < n:
        # Pick starting point of different subarrays of current size
        left_start=0
        while left_start < n:
            # Find ending point of left subarray. mid+1 is starting
            # point of right
            mid = min(left_start + curr_size, n)
            right_end = min(left_start + 2*curr_size, n)
            # Merge Subarrays arr[left_start...mid] & arr[mid+1...right_end]
            iterativeMerge(alist, left_start, mid, right_end)
            left_start += 2*curr_size
        curr_size = 2*curr_size
    return alist
```

Fragmento de Código 5.12. Implementação em Python da função iterativeMergeSort

```
    # Function to merge the two haves alist[0..m] and alist[m..r]
    def  iterativeMerge(alist,l,m,r):
        # create temp arrays
        L = alist[l:m]
        R = alist[m:r]
        print(L)
        print(R)
        # Merge the temp arrays back into alist
        i = 0
        j = 0
        k = l
        while (i < len(L) and j < len(R)):
            if (L[i] <= R[j]):
                alist[k] = L[i]
                i = i + 1
            else:
                alist[k] = R[j]
                j = j + 1
            k = k + 1

        #* Copy the remaining elements of L[], if there are any */
        while (i < len(L)):
            alist[k] = L[i]
            i = i + 1
            k = k + 1

        while (j < len(R)):
            alist[k] = R[j]
            j = j + 1
            k = k + 1
        return alist
```

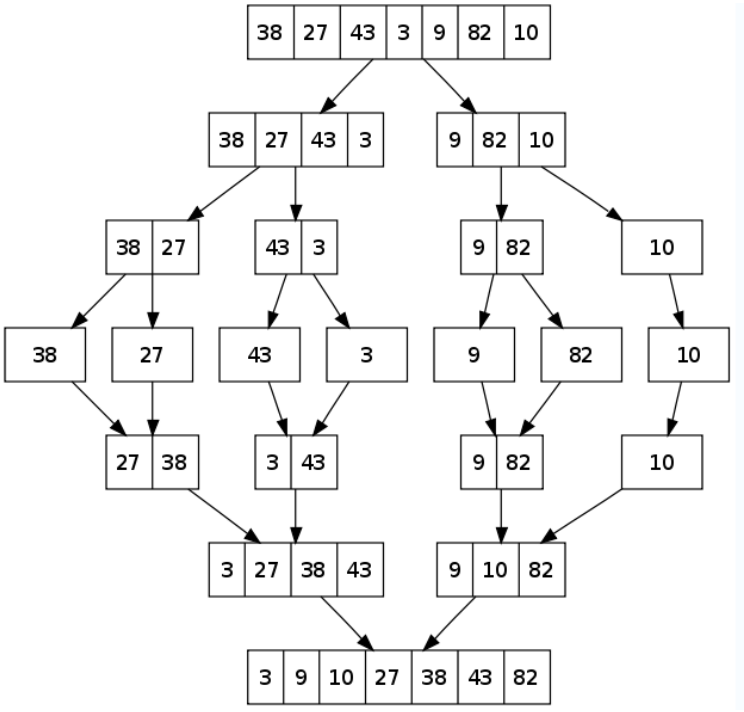
Fragmento de Código 5.13. Implementação em Python da função IterativeMerge

Recursive Merge Sort

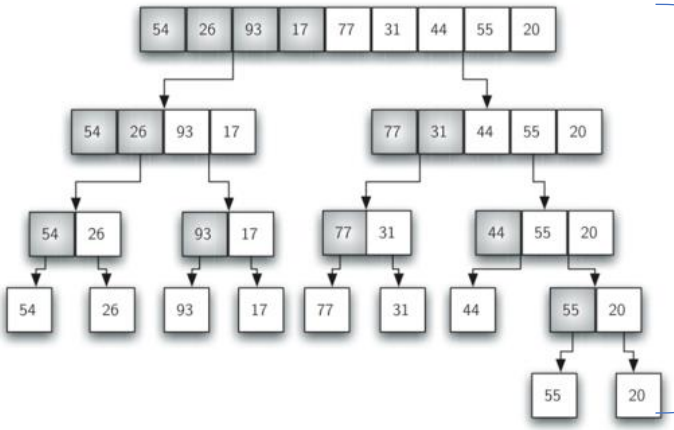
O conceito básico deste método é o mesmo do método anterior, interativeMergeSort, porém, a rotina é recursiva. A ideia básica é, divida o vetor em dois vetores, e para cada vetor resultante, repita o procedimento de divisão em dois vetores. Quando todos os vetores tiverem tamanho uma unidade, este processo de divisão é terminado.

Em seguida. Executamos um merge com pares de vetores de tamanho uma unidade, gerando vários vetores de tamanho dois. Repete-se o processo anterior até que se execute um último merge com os dois últimos vetores.

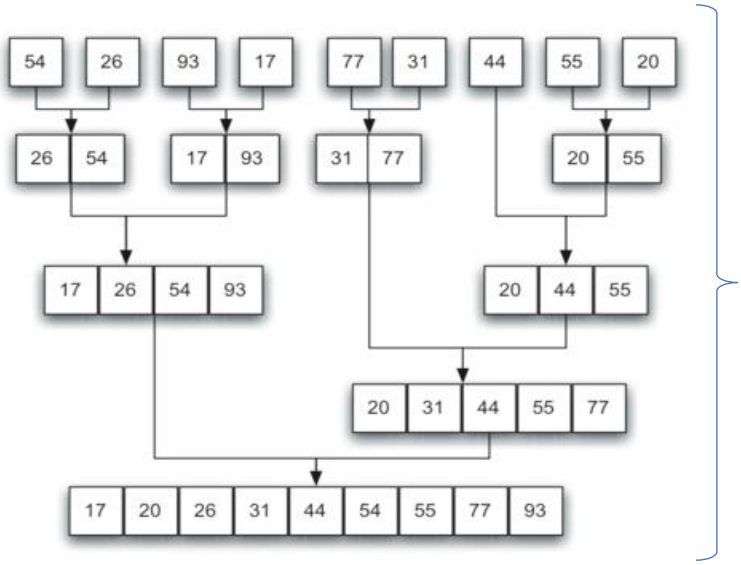
Ordenar o conjunto de chaves 38, 27, 43, 3, 9, 82 e 10 utilizando o método de RecMergeSort. A figura a seguir ilustra o procedimento de subdivisão do vetor inicial até obtemos n vetores de tamanho uma unidade e, depois, o processo inverso, onde se mergeiam os vetores até se chegar no vetor ordenado.



Outro exemplo



Ativação recursiva do MergeSort no vetor à esquerda e direita



Ativação do Merge no vetor gerado pela recursividade anterior

O procedimento recursiveMergeSort está implementado no Fragmento de Código a 5.11.

```
def recursiveMergeSort(alist):
    N = len(alist)
    if len(alist)>1:
        mid = N // 2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]
        lefthalf=recursiveMergeSort(lefthalf)
        righthalf=recursiveMergeSort(righthalf)
        alist=merge(lefthalf+righthalf,mid)
    return alist
```

Fragmento de Código 5.15. Implementação em PYTHON da função recursiveMergeSort