

# MergeSort + Heap

# MergeSort definition

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The `merge()` function is used for merging two halves. The `merge(arr, l, m, r)` is key process that assumes that `arr[l..m]` and `arr[m+1..r]` are sorted and merges the two sorted sub-arrays into one.

- MergeSort is a "stable" sort.
- $O(n \times \log n)$  worst-case, best-case and average-case

# Stable sort

Stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). That is, a sorting algorithm is stable if whenever there are two records  $R$  and  $S$  with the same key and with  $R$  appearing before  $S$  in the original list,  $R$  will appear before  $S$  in the sorted list

Basically sorting if two object are equal in a sense of `compareTo()` method, they remain in the same order as original list even after the sort is done.

# Pseudocode

```
def mergeSort(arr):  
    if len(arr) > 1:  
        mid = len(arr) // 2    # Finding the mid of the array (integer division)  
        L = arr[:mid]          # Dividing the array elements  
        R = arr[mid:]          # into 2 halves  
  
        mergeSort(L)           # Sorting the first half  
        mergeSort(R)           # Sorting the second half  
  
    i = j = k = 0  
  
    while i < len(L) and j < len(R):  
        if L[i] < R[j]:        # If left array is less than right array then use left array  
            arr[k] = L[i]  
            i+=1                # Don't forget to increment index of left array index  
        else:  
            arr[k] = R[j]      # If right array is less than right array then use right array  
            j+=1                # Don't forget to increment index of right array index  
            k+=1                # Regardless, increment main array's index  
  
    while i < len(L):           # Checking if any element is remaining in left array  
        arr[k] = L[i]  
        i+=1                    # Increment left array's index  
        k+=1                    # Increment main array's index  
  
    while j < len(R):           # Checking if any element is remaining in right array  
        arr[k] = R[j]  
        j+=1                    # Increment right array's index  
        k+=1                    # Increment main array
```

# Exercise

Let's try to sort `[9, 8, 7, 6, 5, 4, 3, 2, 1]` using MergeSort

# Solution

```
[9, 8, 7, 6, 5, 4, 3, 2, 1] // initial state
[9, 8, 7, 6] [5, 4, 3, 2, 1] // split +1
[9, 8] [7, 6] [5, 4] [3, 2, 1] // split +2
[9] [8] [7] [6] [5] [4] [3] [2, 1] // split +3
[9] [8] [7] [6] [5] [4] [3] [2] [1] // split +4
[9] [8] [7] [6] [5] [4] [3] [1, 2] // merge -4
[8, 9] [6, 7] [4, 5] [1, 2, 3] // merge -3
[6, 7, 8, 9] [1, 2, 3, 4, 5] // merge -2
[1, 2, 3, 4, 5, 6, 7, 8, 9] // merge -1
```

# Heap

A binary heap is a binary tree where:

- "MinHeap" smallest value is always at the top
- "MaxHeap" largest value is always at the top

# Heap Construction

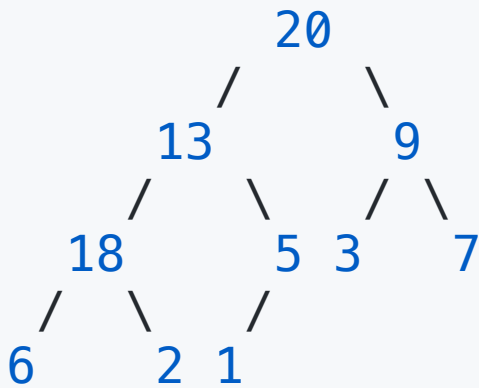
Most efficient implementation of heap uses an array where:

- given index  $i$ , to get:
  - left child index:  $2 \times i + 1$
  - right child index:  $2 \times i + 2$
- parent index:  $\lfloor \frac{i-1}{2} \rfloor$



# Exercise

Represent this tree "MaxHeap" into an array of size 10



# Solution

```
[20, 13, 9, 8, 5, 3, 7, 6, 2, 1]
```

# Pseudocode for "MaxHeap"

```
# 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:
        # Swap
        swap(arr, i, largest)

        # Heapify the root.
        heapify(arr, n, largest)
```

# Pseudocode for "HeapSort"

```
# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)

    # Build a maxheap
    for i in range(n, -1, -1):    # start from n, go until -1 and increment by -1
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):    # start from n-1, go until 0 and increment by -1
        # Swap
        swap(arr, i, 0)
        heapify(arr, i, 0)
```

# Lab Exercise

## MergeSort