Name: Solutions

Login:

Signature:

ECE 368 Spring 2016

Homework 4

1. Heap Sort (30pts)

a) The time complexity for operations on binary min-heaps is the following:

	front	arbitrary	back
insert	$O(\ln(n))^*$	$O(1)$	$O(1)$
access	$O(1)$	$O(n)$	$O(n)$
delete	$O(\ln(n))$	$O(n)$	$O(n)$

Justify the time complexity for each entry. **(10pts)**

- **Insert:**
  - Front:
    - Min Object is Inserted: Needs to be swapped $log(n)$ times until reaching the top of the binary heap.
  - Arbitrary:
    - On average, it will take $O(1)$ moves because half of the entries are leaf nodes.
    - $\frac{1}{n}\sum_0^h (h-k)2^k = \frac{2^{k+1}-h-2}{n} = \frac{n-h-2}{n} = O(1)$
  - Back:
    - Max Object is inserted: It is inserted just to the end of the array in O(1).
- **Access:**
  - Front: Only the root node needs to be checked.
  - Arbitrary: All the leaf nodes need to be checked in a BFS traversal O(n).
  - Back (Max Number): All the leaf nodes need to be checked in a BFS traversal O(n).
- **Delete:**
  - Front: Remove the top element and then replace element with its greater child. Worse case: traverse all the way to the leaf -$log(n)$-1 times = $O(lgn)$.
  - Arbitrary/Back: Access the elements first $O(n)$, then reheapify $O(logn)$, O(n + logn) = O(n)

b) There are the following two strategies for pop using heaps:

1. Remove the top and heapify

2. Replace the top with the last element in an array implementation and then percolate it down.

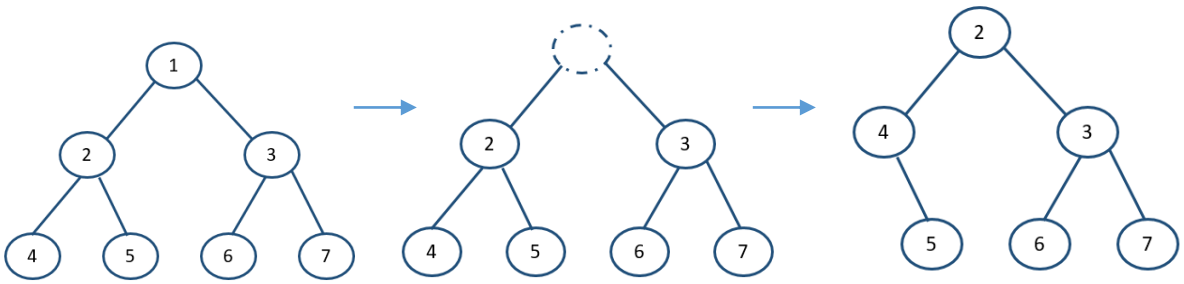
Assume we start with a binary min-heap that is also a complete tree. Which of these two strategies does not maintain the completeness of the tree? Give a counter example if the strategy that does not maintain completeness and provide an explanatory proof if it does. **(10pts)**

A complete binary tree has all the nodes filled top to bottom and left to right in the last level.

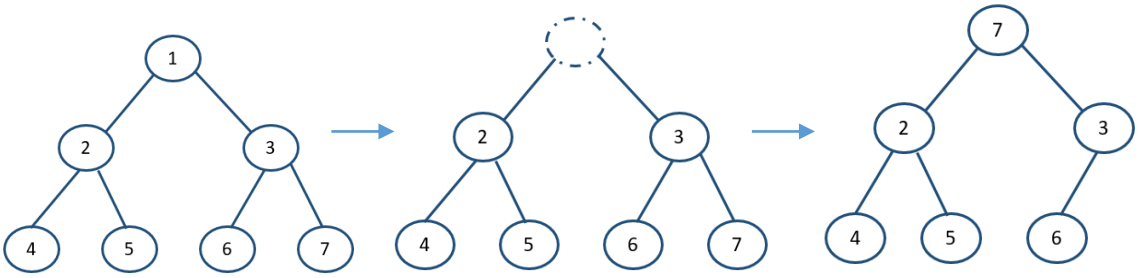
1. Strategy 1 fails to keep completeness of the tree if left child at any level is smaller than right child.

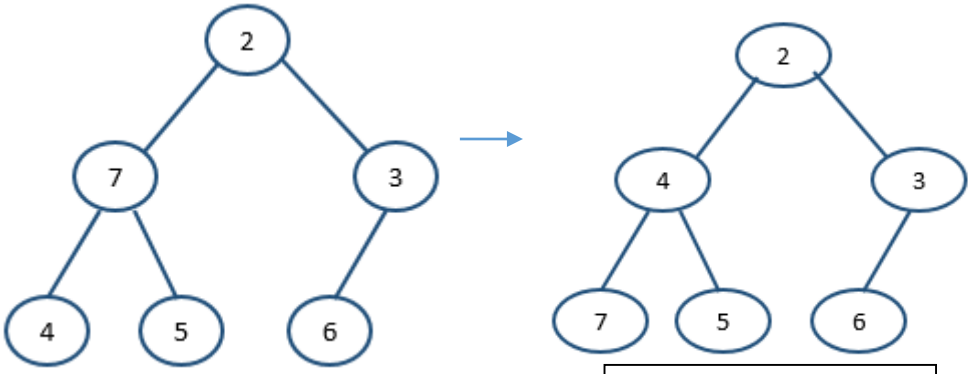
   Example:

Incomplete Binary Tree.



2. Strategy 2 maintains completeness of the tree because it will always remove the last element from the right at bottom level.

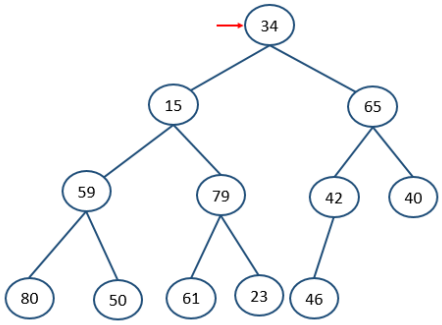


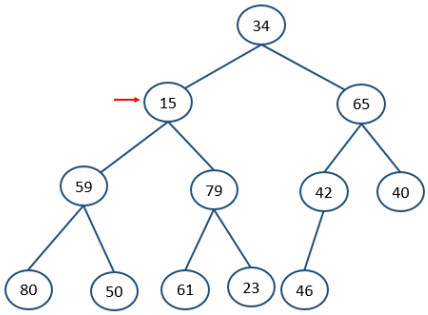


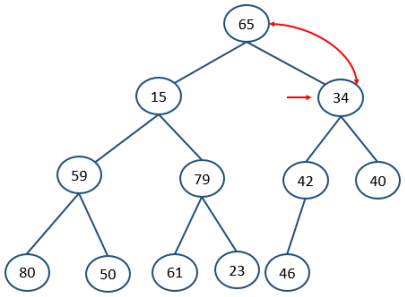
Complete Binary Tree.

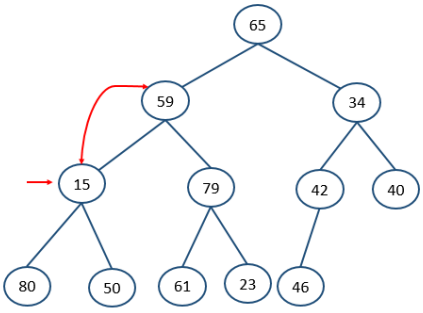
c) Sort the following 12 entries using heap sort. Show each step of heap sort using either tree or array. **(10pts)**

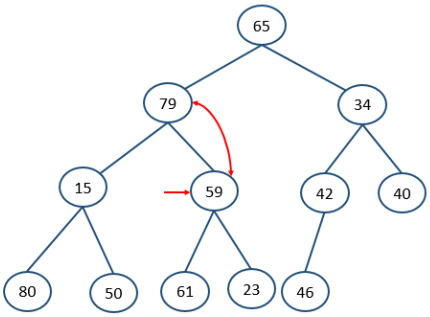
34	15	65	59	79	42	40	80	50	61	23	46

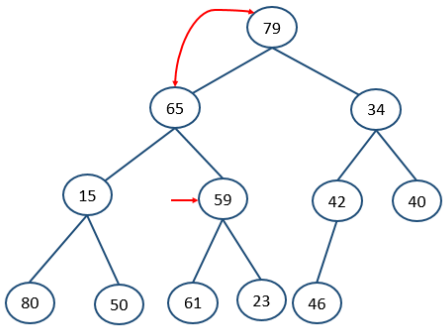


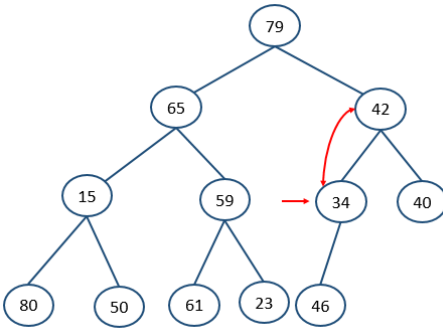


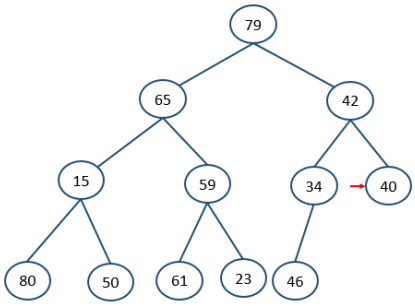


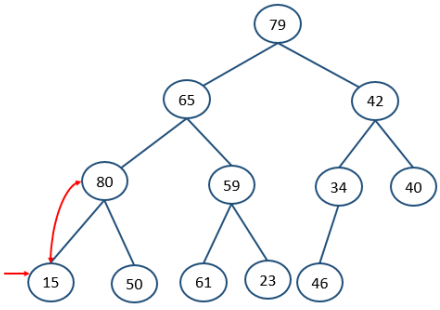


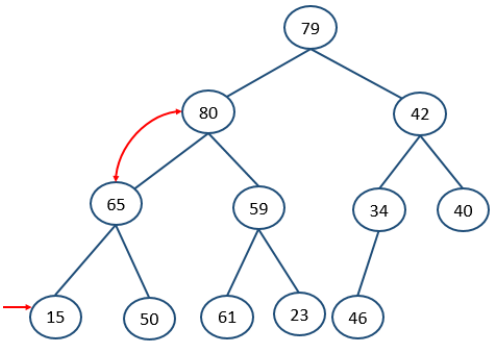


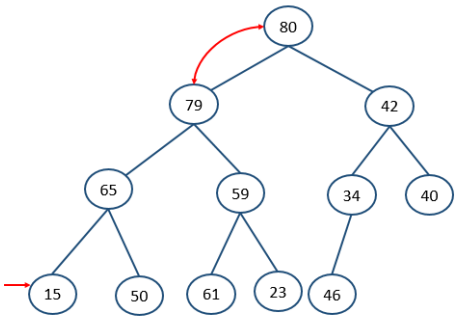


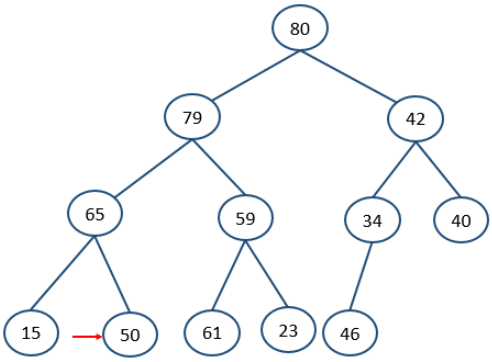


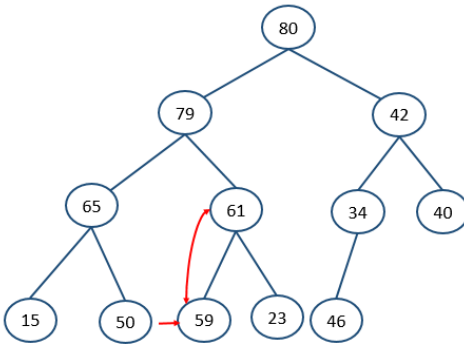


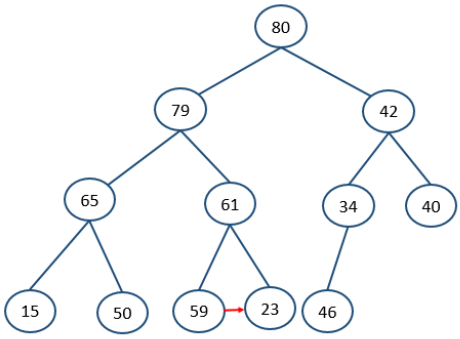


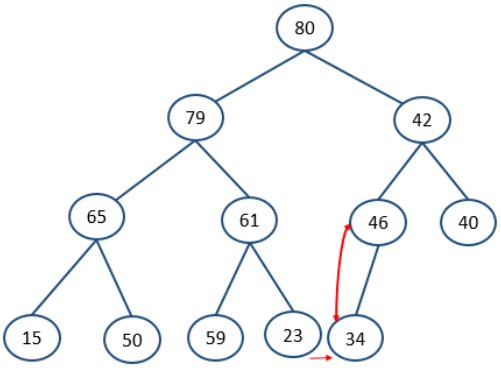


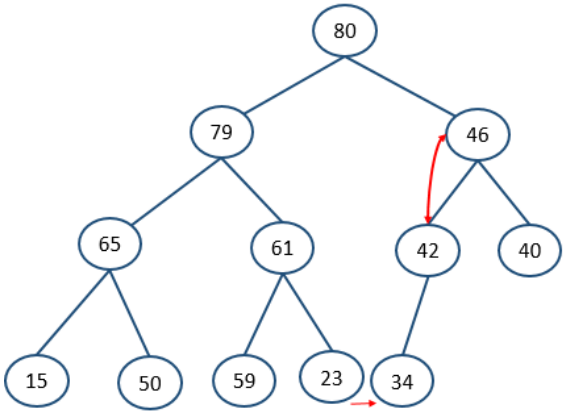






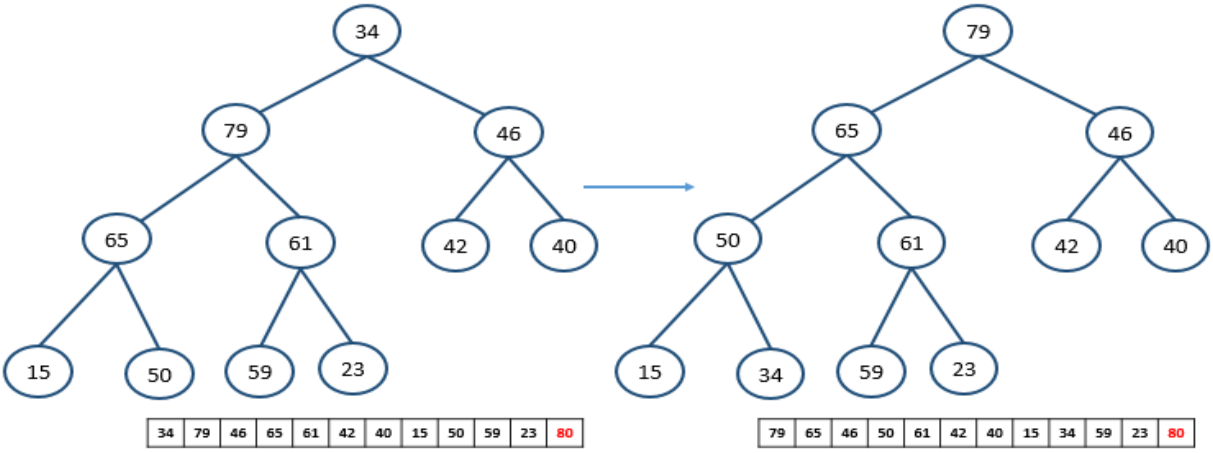


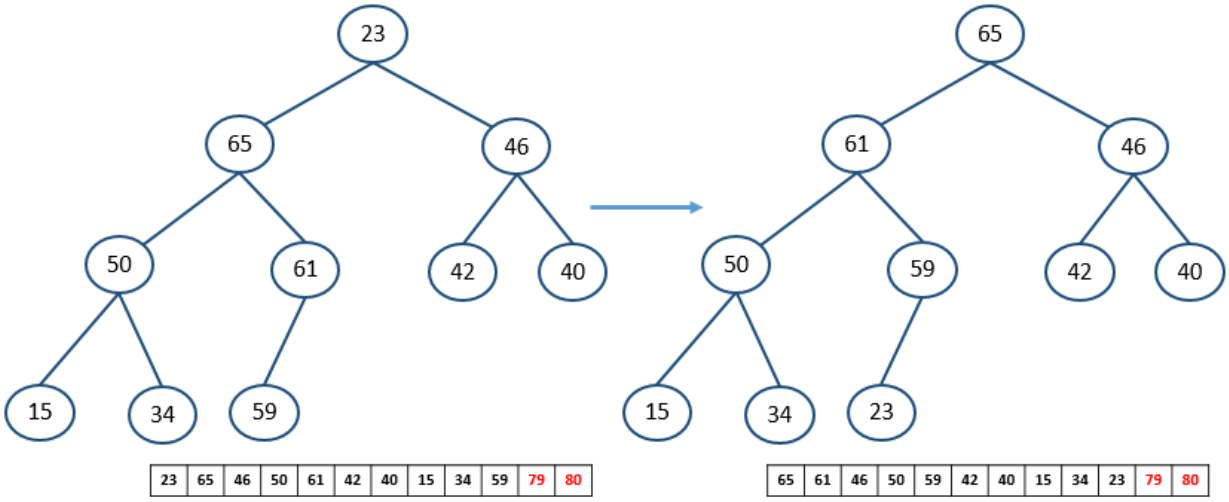


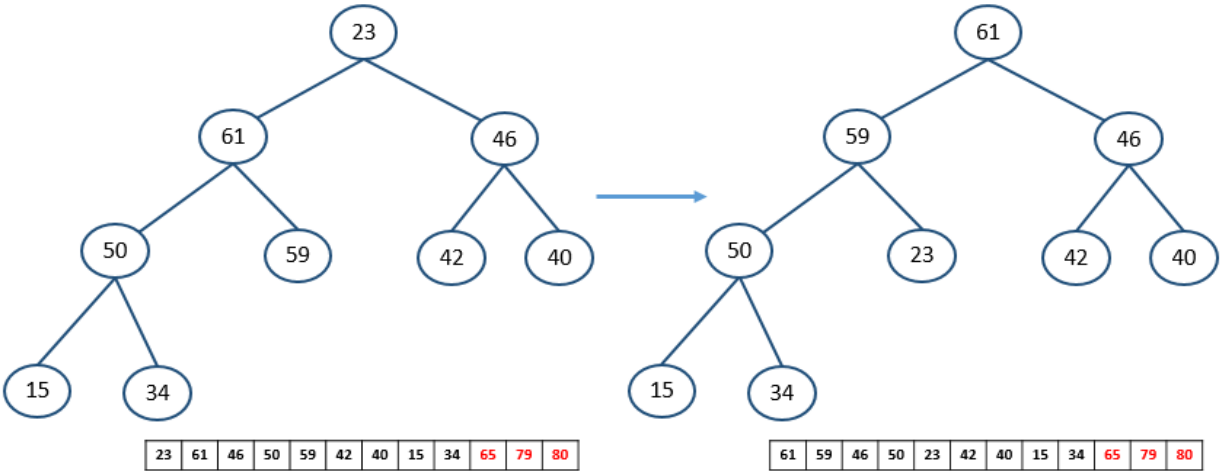


80	79	46	65	61	42	40	15	50	59	23	34

Max Heap but not sorted.
-Now extract the max and put it at the end of the array.
-Put last element at the top and heapify down (swap it with max child recursively).
-New heap has length N-1

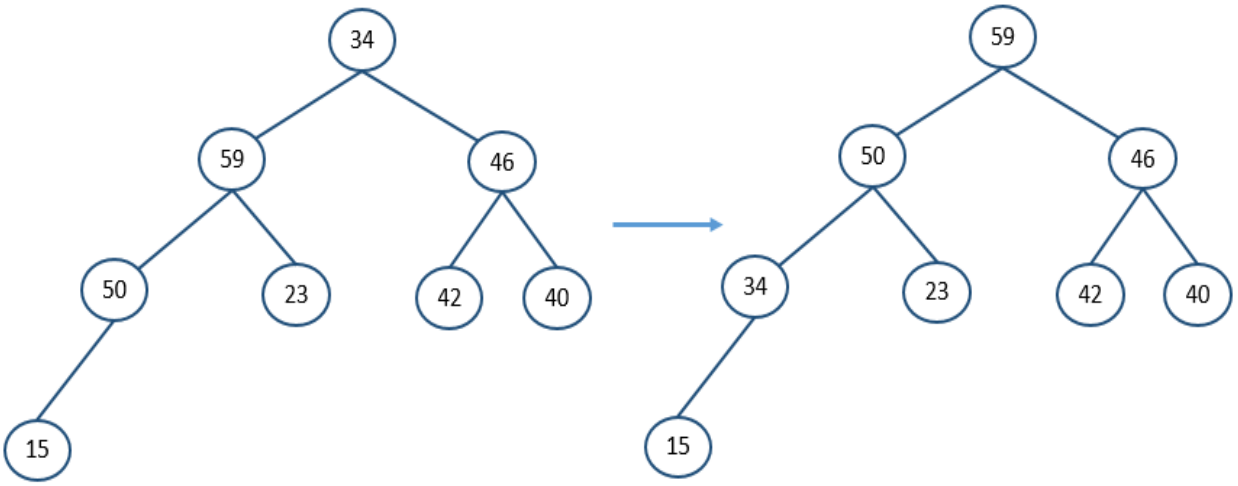

34
79
46
65
61
42
40
15
50
59
23
34 79 46 65 61 42 40 15 50 59 23 80
79
65
46
50
61
42
40
15
34
59
23
79 65 46 50 61 42 40 15 34 59 23 80


23
65
46
50
61
42
40
15
34
59
23 65 46 50 61 42 40 15 34 59 79 80
65
61
46
50
59
42
40
15
34
23
65 61 46 50 59 42 40 15 34 23 79 80



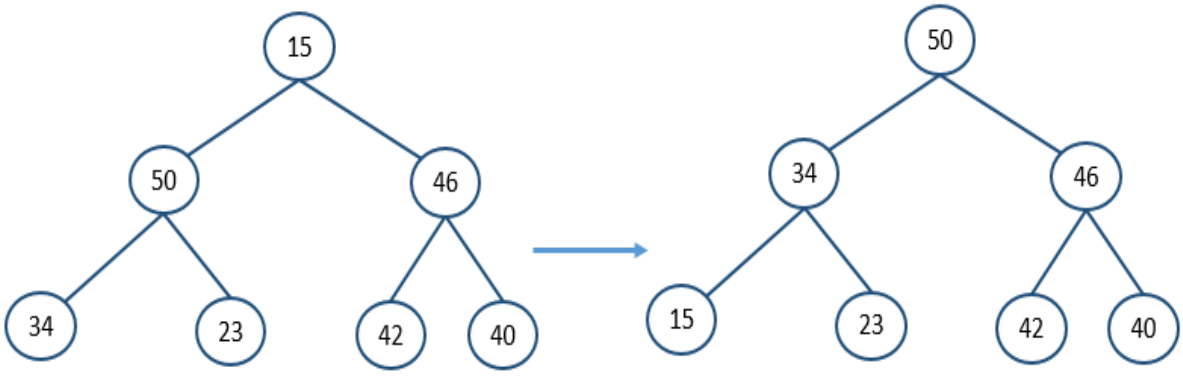
23	61	46	50	59	42	40	15	34	65	79	80

61	59	46	50	23	42	40	15	34	65	79	80



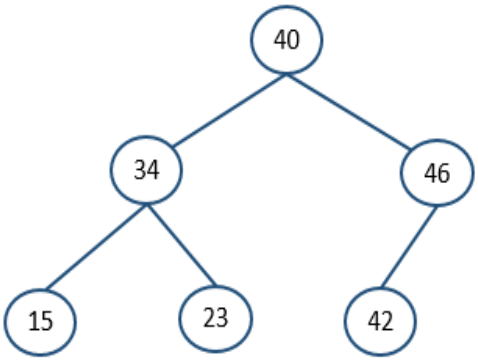
34	59	46	50	23	42	40	15	61	65	79	80

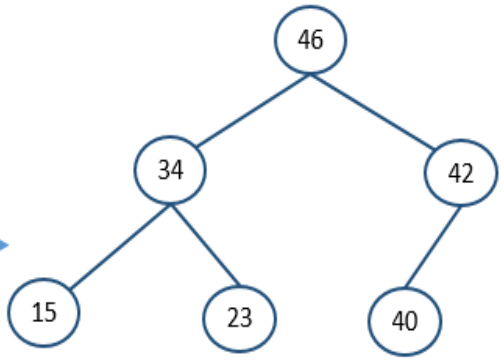
59	50	46	34	23	42	40	15	61	65	79	80



15	50	46	34	23	42	40	59	61	65	79	80

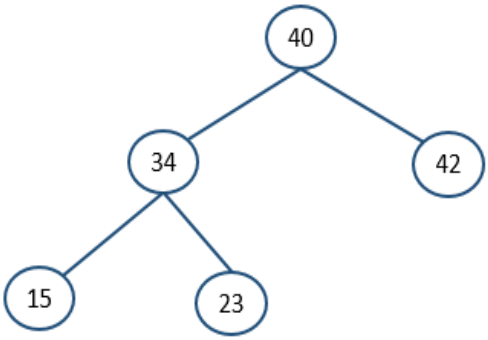
50	34	46	15	23	42	40	59	61	65	79	80

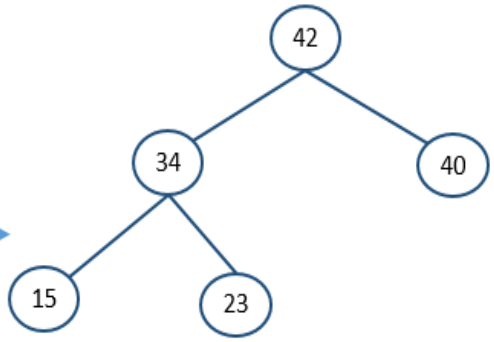




40	34	46	15	23	42	50	59	61	65	79	80

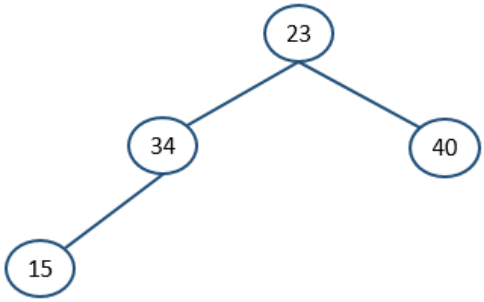
46	34	42	15	23	40	50	59	61	65	79	80

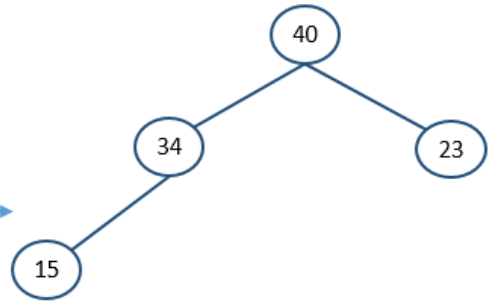




40	34	43	15	23	46	50	59	61	65	79	80

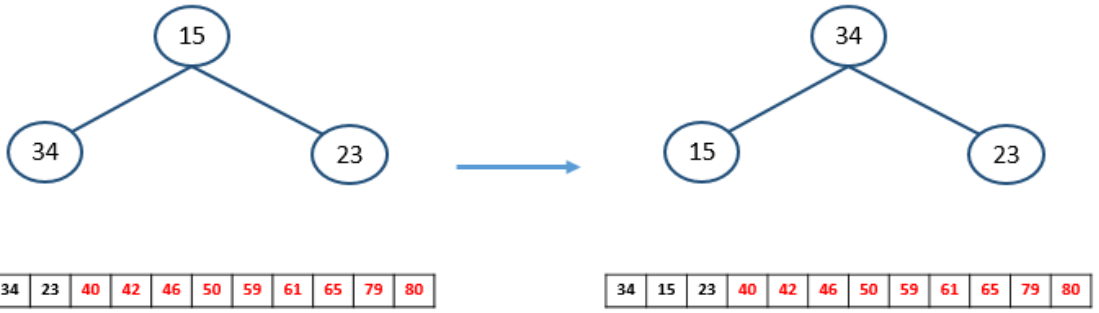
42	34	40	15	23	46	50	59	61	65	79	80



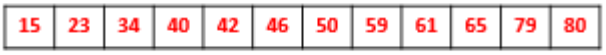


23	34	40	15	42	46	50	59	61	65	79	80

40	34	23	15	42	46	50	59	61	65	79	80



Sorted Array:



2. Insertion Sort vs Merge Sort (10pts)

Consider a category of unsorted arrays where the number of inversions grows linearly with the size of the array. In other words, the number of inversions is O(n). Which one is better, insertion sort or merge sort? Explain the reason.

From the lecture slides, the average case for insertion sort is O(d+n). In this case, d=O(n). Merge sort **always needs O(nlogn)** operations. Therefore, insertion sort is better because we are talking about the **best case** for insertion sort. **O(n).**

The following table summarizes the run-times of insertion sort

Case	Run Time	Comments
Worst	$\Theta(n^2)$	Reverse sorted
Average	$O(d+n)$	Slow if $d=\omega(n)$
Best	$\Theta(n)$	Very few inversions: $d=O(n)$

3. Programming Questions (60 Pts):

a) Given the "hw4.c" file, write the following functions:

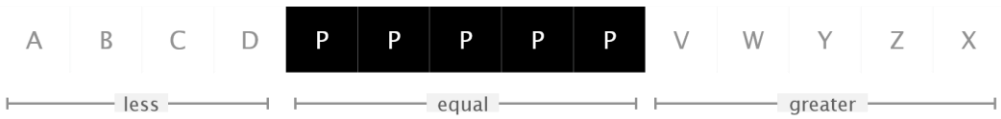
InsertionSort (10 Pts)

QuickSort1 (10 Pts)

QuickSort2 (10 Pts)

QuickSort3P (20 Pts)

- Notes:
  - For QuickSort1, make pivot the first element of subarray (Array[first]).
  - For QuickSort2, make pivot the median between the <u>first, middle, and last</u> element from the array (Array[first], Array[floor((first+last)/2)], Array[last].
  - For QuickSort3P, implement 3 way partitioning quicksort which partitions the array into 3 parts (less than, equal to and greater than the pivot):



  - Your files will be graded by inputting an Array stored in file named "Array.txt" or "Array2.txt" and checking the results in a file named "Sorted.txt". **Changing this names will jeopardize your grading**.

  - You need to submit only one file named "hw4.c".

b) Algorithm Complexity **(10 Pts)** **WILL DEPEND ON IMPLEMENTATION. CODE SOLUTION WILL BE POSTED SOON.**

For the input file "Array2.txt", what are the total numbers of comparisons and moves for:

Algorithm	Comparisons	Moves
InsertionSort		
QuickSort1		
QuickSort2		
QuickSort3P		

- Notes:
  - Whenever you use the "=" operator, add +1 move.
  - Whenever you use the ">, <, ==, <=, >=" operators or loop through a **while/for** loop, add +1 comparison.
  - Answers may vary depending on your implementation.

c) Extra Credit **(5 Pts)**

For the input file "Array1.txt", what are the total numbers of comparisons and moves for:

Algorithm	Comparisons	Moves
InsertionSort		
QuickSort1		
QuickSort2		
QuickSort3P		