**Question 1. Merge Sort**

a. Merge sort on: [38, 27, 43, 3, 9, 82, 10]

MergeSort(array):

Splitting the array into two halves

MergeSort(left half)

MergeSort(right half)

Merge(left half, right half)

MergeSort([38, 27, 43, 3, 9, 82, 10])

Two halves: [38, 27, 43, 3] [9, 82, 10]

MergeSort([38, 27, 43, 3])

Two halves: [38, 27] [43, 3]

MergeSort([38, 27])

Two halves: [38] [27]

MergeSort([38]) Do nothing, since 1 element

MergeSort([27]) Do nothing

Merge([38],[27]) [27, 38]

MergeSort([43, 3])

Two halves: [43, 3]

MergeSort([43])

MergeSort([3])

Merge([43], [3]) [3, 43]

Merge([27, 38], [3, 43]) [3, 27, 38, 43]

MergeSort([9, 82, 10]) , so 2 numbers in left half

Two halves: [9, 82] [10]

MergeSort([9, 82])

Two halves: [9] [82]

MergeSort([9])

MergeSort([82])

Merge([9],[82]) [9, 82]

MergeSort([10])

Merge([9, 82], 10) [9, 10, 82]

Merge([3, 27, 38, 43], [9, 10, 82]) [3, 9, 10, 27, 38, 43, 82]

b. **Major Advantage**: Stability, where if two elements were in a certain order before the sort, that order is still maintained after the sort.

[3, 4, 3, 5, 6, 9, 8] [3, 3, 4, 5, 6, 8, 9]

c. **Major Disadvantage**: Space Complexity, since the Merge algorithm requires the use of a temporary array, which is a copy of the original array. Although there are variations that sort in-place, they can be quite complex.

Question 2: Quick Sort

a. **The split be most extremely skewed when**: the array is sorted in non-descending order where the pivot is not repeated.

[1, 2, 2, 3, 4, 5, 6, 7] Can’t find that is at least the pivot, is the pivot itself, so partition will return = 0.

Result in the first split where the left array has no element, and the right array has all the elements on the right of the pivot.

b. **Apply the Quick Sort Algorithm to**

[E, X, A, M, P, L, E]

QuickSort(array):

S Partition(array) // S is a split position:

While i < j

Find i from the left such that array[i] p

Find j from the right such that array[j] p

Swap (array[i], array[j])

Undo last swap

Swap(A[l], A[j]) // Swap the pivot with j. So j becomes the pivot

QuickSort(left half up to S - 1)

QuickSort(right half from S + 1)

QuickSort([E, X, A, M, P, L, E])

S Partition([**E**, X, A, M, P, L, E]):

Pivot:

at X, at right of

Find such that element at i E

at X

Find such that element at j E

at E

Swap X and E

[**E**, E, A, M, P, L, X]

Find such that element at i E

at M

Find such that element at i

at A

Swap M and A

[**E**, E, M, A, P, L, X]

Since i j, break the loop

Undo swap:

[**E**, E, A, M, P, L, X] j = 2

Swap pivot with array[j]

[A, E, **E**, M, P, L, X]

Return 2

QuickSort([A, E])

S Partition([**A**, E])

: 0 1 (found E > A)

: 2 (found A A)

Swap elements at 0 and 1

Undo the swap

QuickSort([])

QuickSort([**E**])

QuickSort([M, P, L, X])

S Partition([**M**, P, L, X])

: where P **M**

where L **M**

Swap elements 1 and 2

[**M**, L, P, X]

where P M

where LM

Swap elements 2 and 1

[**M**, P, L, X]

Undo the last swap

[**M**, L, P, X]

Swap pivot with element 1

[L, **M**, P, X]

Return 1

QuickSort([**L**])

QuickSort([**P**,X])

[A, E, E, L, M, P, X]

**Question 3: Complete binary tree and Heap Sort**

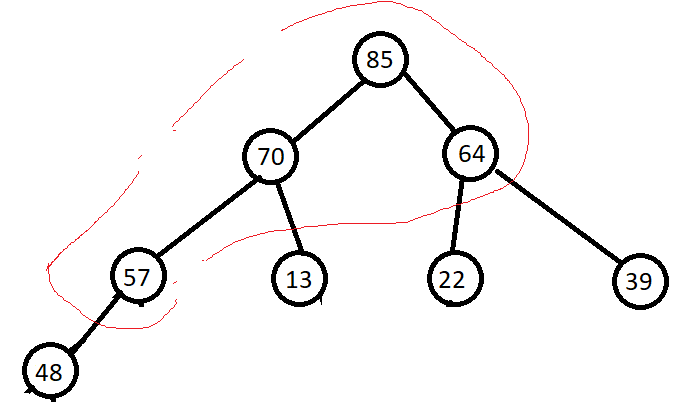
1. **Which the following are true:**

A complete binary tree is a tree filled from top to bottom, left to right

1. Cannot be true, because all nodes at level must have two children
2. True, since we are filling from left to right, the only way for a node at level to have one child is for all nodes left of it to be fully filled up (two children each)
3. Cannot be true, since we are filling from left to right, the left node must always be filled up first. Must be a left child, not right
4. True, since for a node with index , at height (starting from 0 at the root), then its left child will be and the right will be .
5. **Which of the following are complete binary trees?**
6. Yes
7. No, since E has to parents not a binary tree
8. No, since I has a child that is the right child, and C does not have two children
9. No, C has a right child without the left child.
10. **Perform Heap Sort on (non-heap array):**

[13, 57, 39, 85, 70, 22, 64, 48]

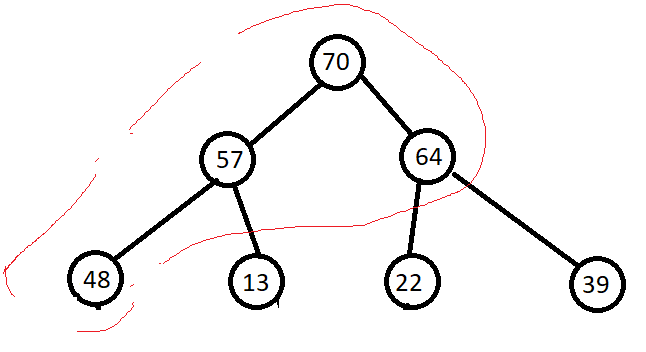
Convert from Complete Binary Tree to Heap



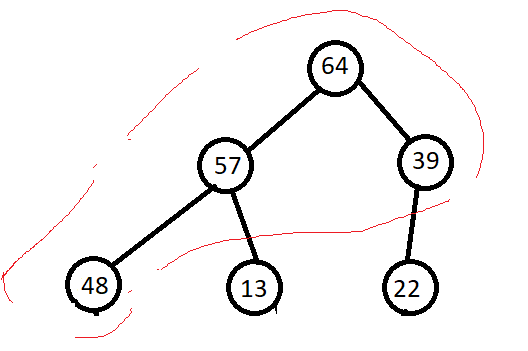
[85, 70, 64, 57, 13, 22, 39, 48]

Repeatedly perform Maximum Key Deletion

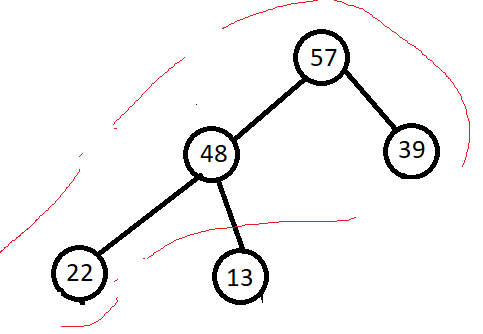
Delete the root 85, make the last element the root, then re-heapify



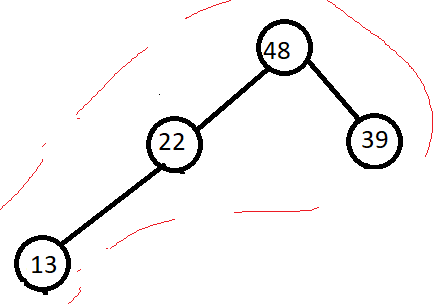
Delete the root 70, make the last element the root, then re-heapify



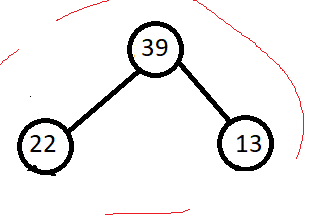
Delete root 64, re-heapify



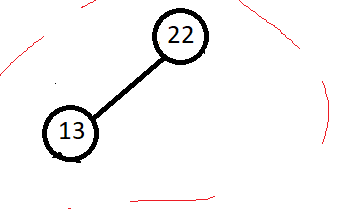
Delete root 57, re-heapify



Delete 48, re-heapify



Delete 39, re-heapify



Delete 22, re-heapify



Result:

