Question 1:

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

Merge(array left, array right): merge two sorted arrays together

MergeSort:

Split the array into two halves

MergeSort(first half)

MergeSort(second half)

Merge(first half, second half)

[38, 27, 43, 3, 9, 82, 10]

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

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

MergeSort([38, 27]):

MergeSort([38])

MergeSort([27])

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

MergeSort([43, 3])

MergeSort([43])

MergeSort([3])

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

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

MergeSort([9, 82, 10]):

MergeSort([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]**

1. Major Advantage: **More stable**. If two elements of the same sorting order (same key) will remain in the same order after sorting.
2. Major Disadvantage: **Space complexity**, because the merge algorithm requires extra spaces for the merged array. There are iniplace alternatives, but they are quite complicated.

Quick Sort:

1. Most skewed when: **the array has already been sorted**, the second half is all the elements after the first, since the pivot (first element) has already partitioned the array into two halves, with the right half bigger than it.
2. QuickSort([E,X,A,M,P,L,E])

Partition(array): split the array into two half, where the first half is less than a pivot, second half is greater than a pivot by swapping the elements.

QuickSort(array):

Position Partition(array)

QuickSort(first half)

QuickSort(second half)

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

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

Repeat:

Repeat:

Repeat:

Swap(A[1], A[6])

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

Is ) No, continue

Repeat:

Nope

Yes

Repeat:

Nope

Nope

Nope

Yes

Swap(A[3], A[2]) [**E**, E, M, A, P, L, X]

Is Yes, stop

Swap(A[3], A[2]) [**E**, E, A, M, P, L, X]

Swap(A[0], A[2]) [A, E, **E**, M, P, L, X]

Return

QuickSort([A, E])

S Partition([A,E]) ([A,E], 0)

QuickSort([])

QuickSort([E])

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

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

QuickSort([L])

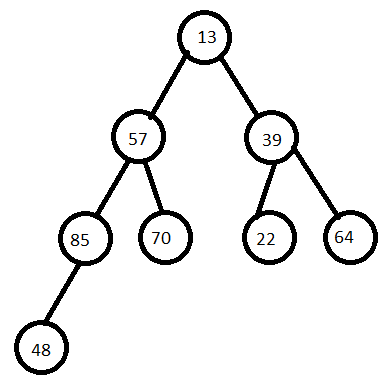
QuickSort([P,X])

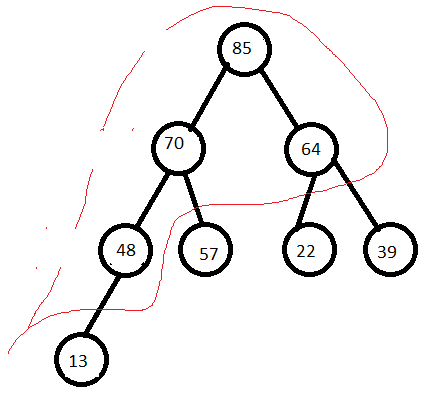
Partiton([P,X]) ([P,X], 0)

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

3. Complete Binary Tree and Heap Sort

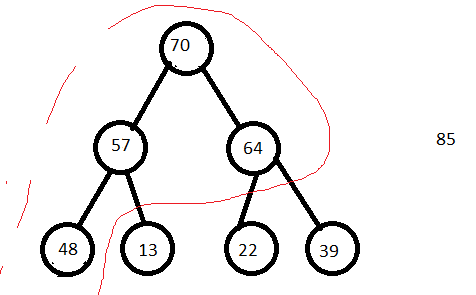
1. The statements, simplified:
2. All nodes two levels above the deepest level must have two children False
3. True
4. Must be left child False
5. True
6. A is a complete binary tree.
7. Perform heap sort on ([13, 57, 39, 85, 70, 22, 64, 48])

  
This is not a heap, because there are nodes whose children are greater than itself. Convert to a heap

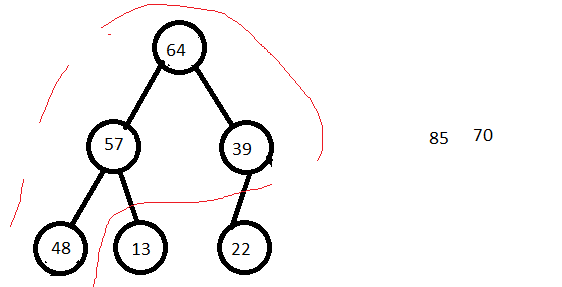


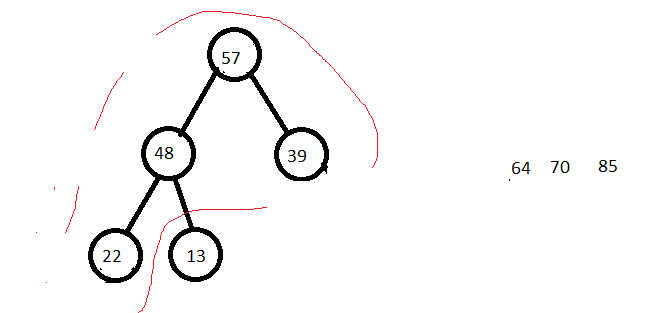
[85, 70, 64, 58, 57, 22, 39, 13]. This is a heap

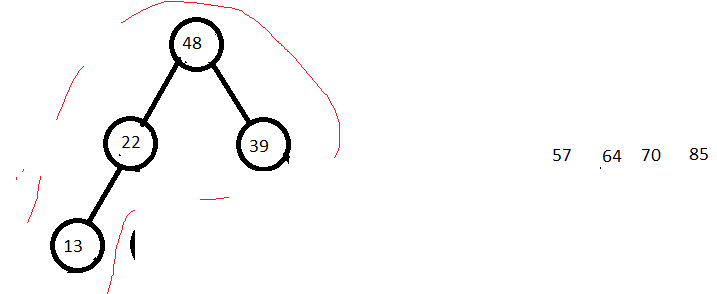
Perform maximum key deletion: 85, then re-heapify



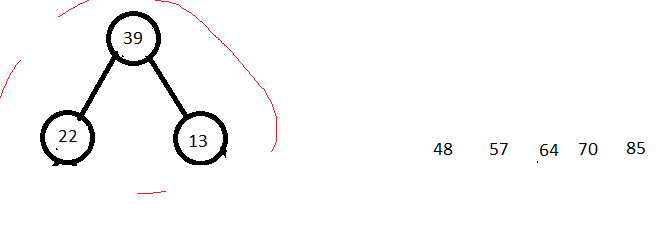
Remove 70, then re-heapify

  
Remove 64 then re-heapify

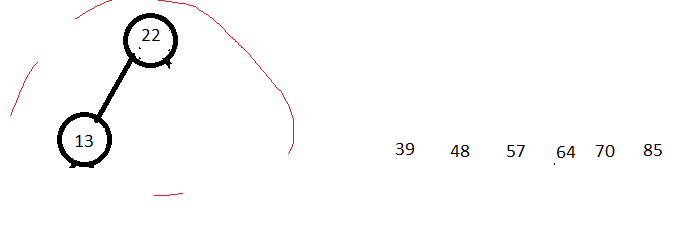
  
Remove 57 then re-heapify



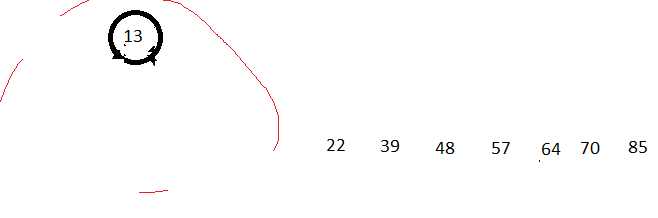
Remove 48 then re-heapify



Remove 39 then re-heapify



Remove 22 then re-heapify



Remove 13 and return

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