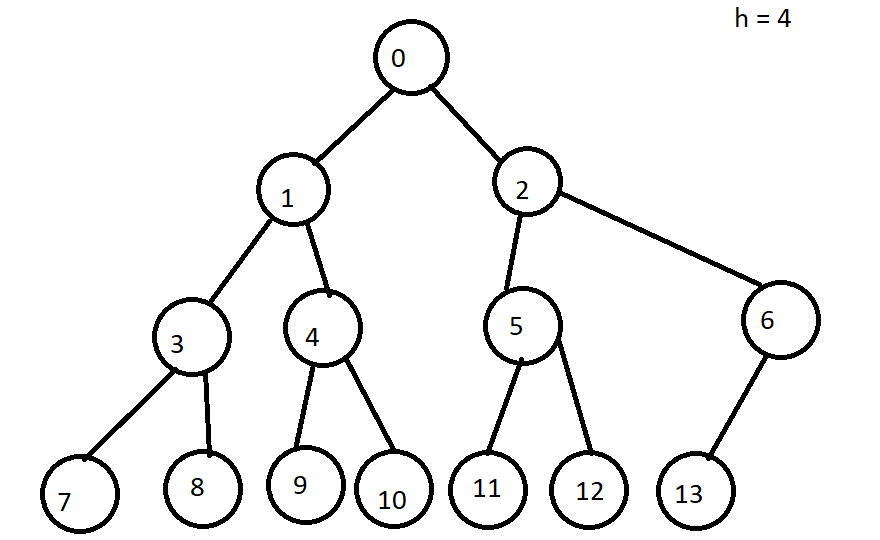
Question 3: Complete binary tree and Heap Sort

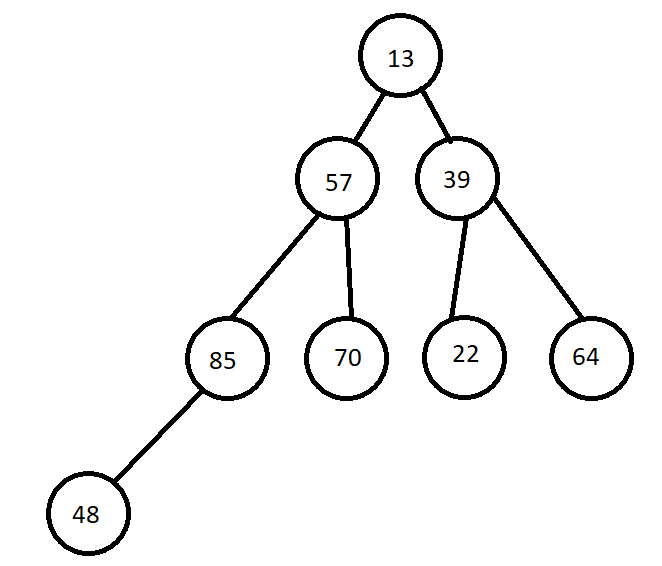
A complete binary tree is a binary tree that is filled from top to bottom then left to right

1. For a complete binary tree of height :
2. **False**. Since to get to height , all nodes on level must be fully filled up, meaning all nodes on level must have two children each
3. **True**. Since by filling from left to right, the only way for a node to have a single child is for all nodes before to be totally filled up (to children each).
4. **False**. Since we are filling from left to right, the left node must always be filled first.
5. **True**. As proven by this diagram.

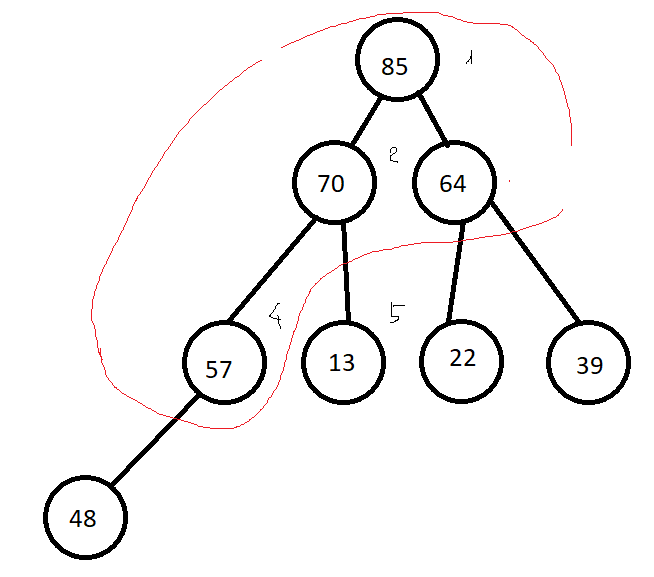


1. A is a complete binary tree.
2. Perform HeapSort([13, 57, 85, 70, 22, 64, 48])

The array can be represented by the complete binary tree

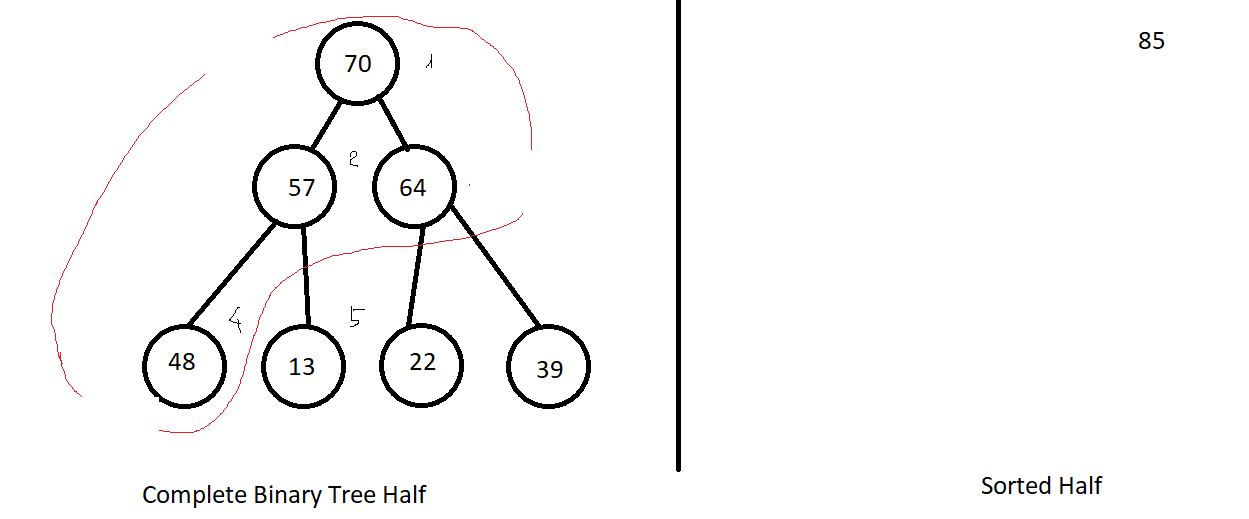


Convert from Complete Binary Tree to Heap, using the HeapBottomUpAlgorithm

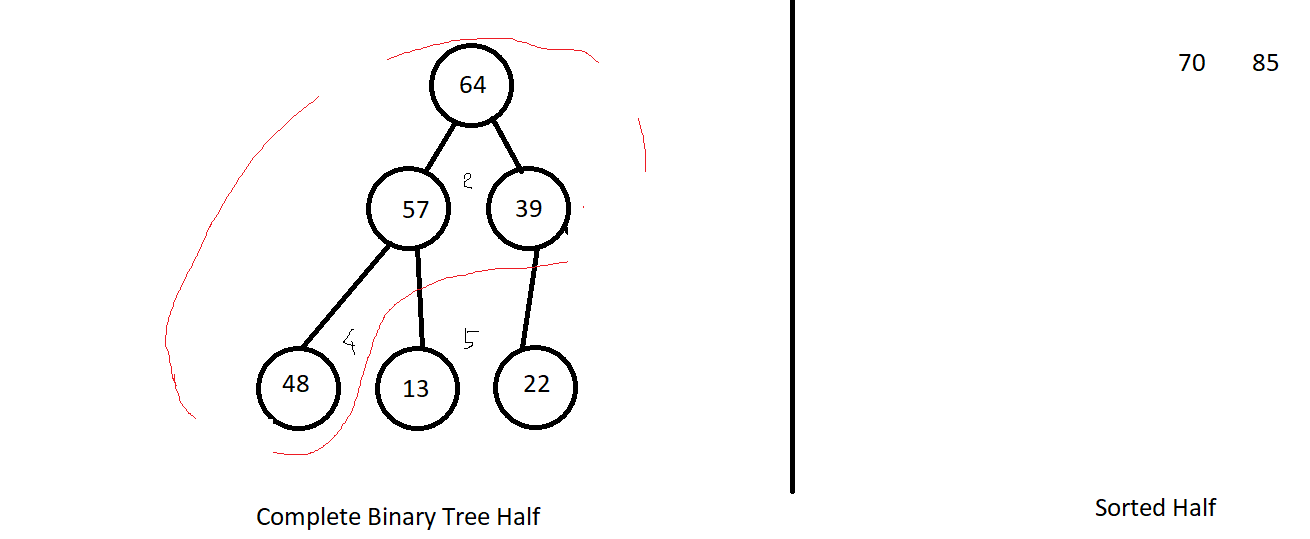


Perform sorting by repeatedly applying MaximumKeyDeletion

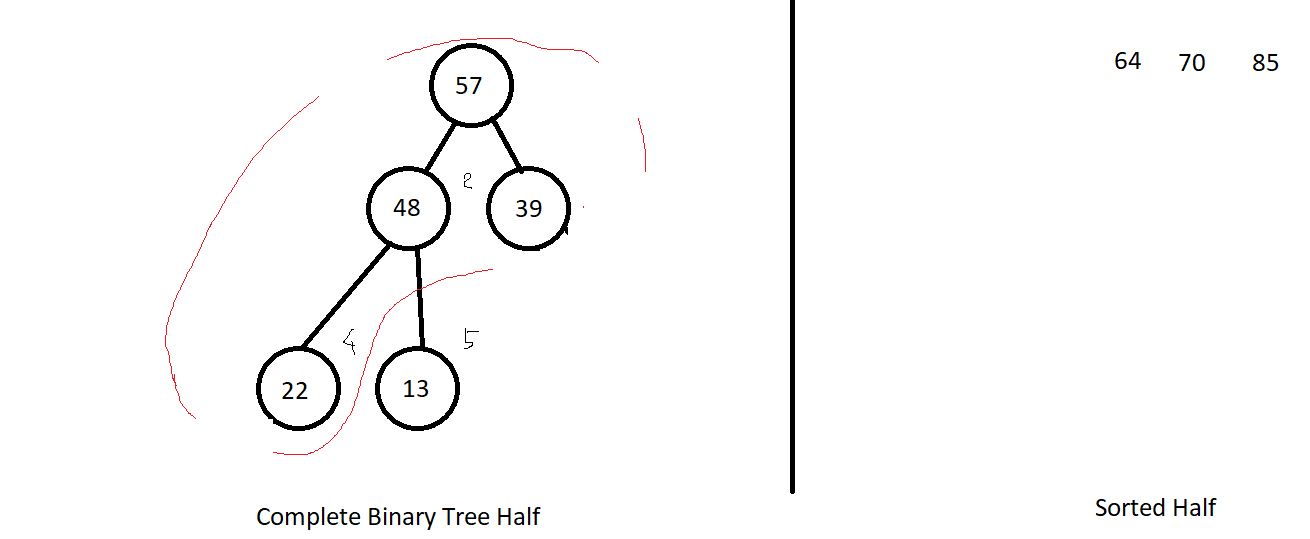
Delete 85 then re-heapify



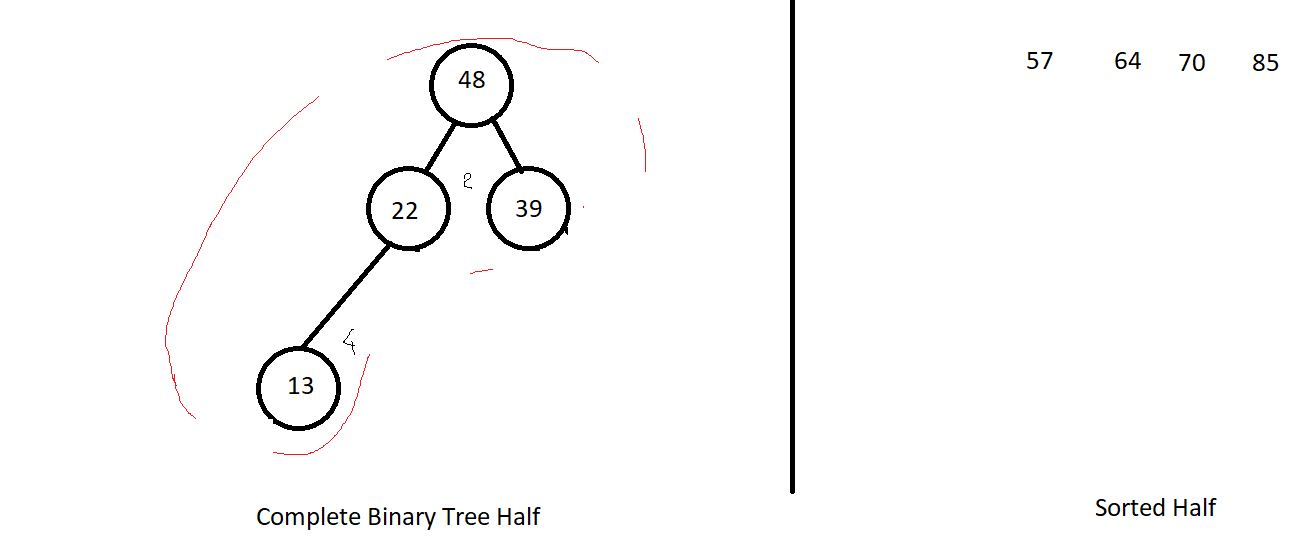
Delete 70 and re-heapify

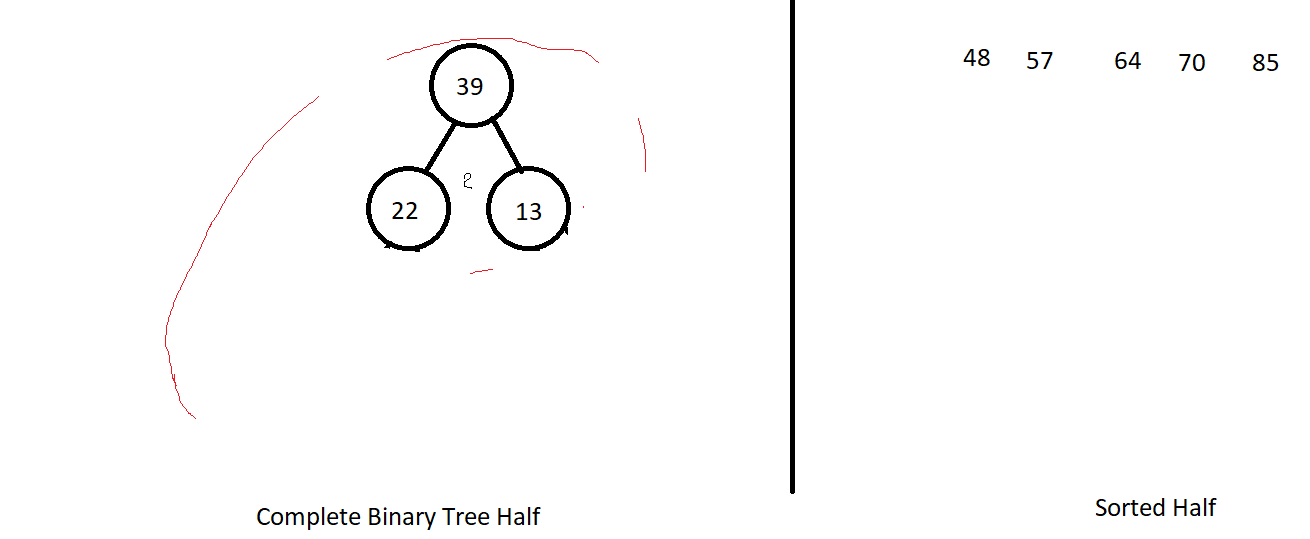


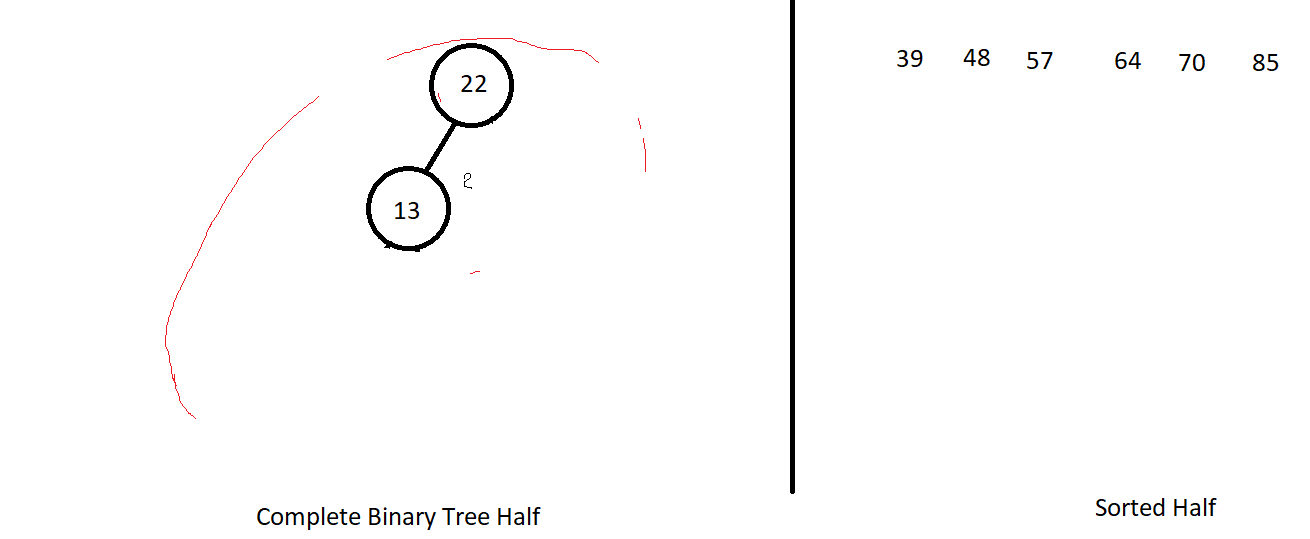
Delete 64 and re-heapify

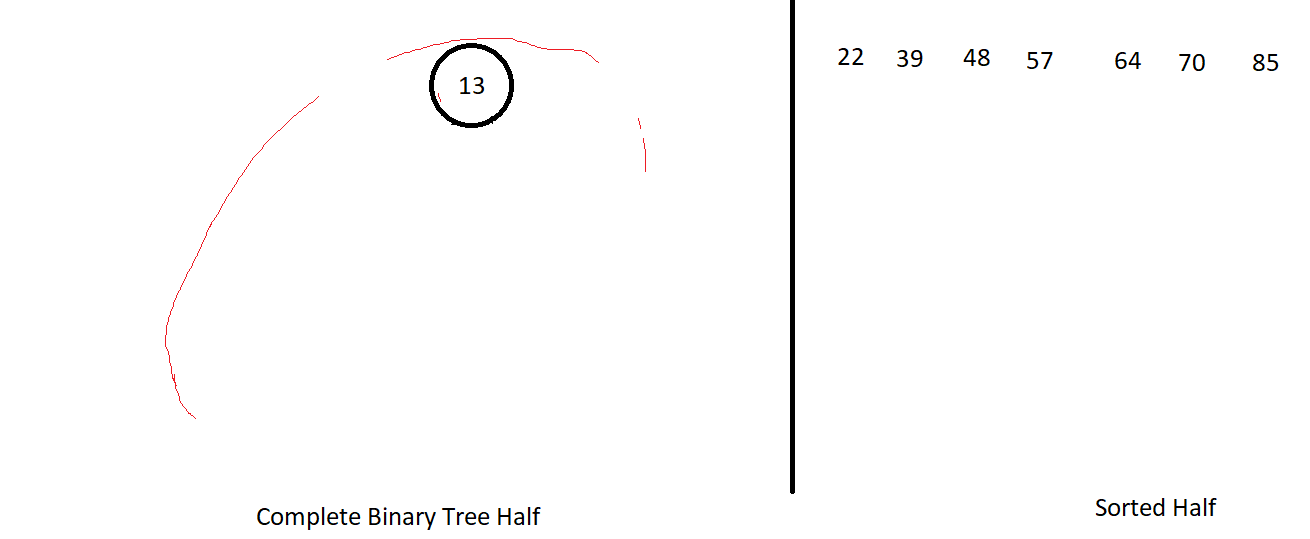


Delete 57 and re-heapify









Question 1: MergeSort

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

MergeSort(array):

Split the array into two halves by the midpoint

MergeSort(left half)

MergeSort(right half)

Merge(left half, right half) // left and right haves are sorted

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]): [27, 38]

Two havles: [38] [27]

MergeSort([38]) [38]

MergeSort([27]) [27]

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

MergeSort([43, 3]):

Two halves: [43] [3]

MergeSort([43]) 43

MergeSort([3]) 3

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

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

MergeSort([9, 82, 10]):

Two halves: [9, 82] [10]

MergeSort([9, 82]) [9, 82]

MergeSort([10]) [10]

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

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

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

1. Major Advantage: Stability. If two elements have the same sorting order before merge sort, they will stay in the same relative order after the merge sort.

E.g. [4, 3, 5, 3, 6] [3, 3, 4, 5, 6]

1. Major Disadvantage: Space Complexity. Since the Merge algorithm requires a temporary array to store the merged array, before overwriting the original one.

**Question 2:**

a. Most skewed when array is already sorted. When looking for , nothing will be less than or equal to the pivot, except itself, meaning the second half of the array will consists of everything after the first element (pivot).

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

QuickSort(array):

Find a position for the pivot using the Partition algorithm:

Repeat until the left cursor is greater than the right

Find such that element at pivot

Find such that element at pivot

Swap elements at and

Undo last swap

Swap pivot with element at (new pivot position)

QuickSort(array[0…pivot – 1])

QuickSort(array[pivot + 1…n – 1])

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

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

Find (X E)

Find (E E)

Swap elements at and : [**E**,E,A,M,P,L,X]

Find (A < E) (M E)

Find (L > E) (P > E) 3 (M > 3) 2 (A E)

Swap elements at 3 and 2: [**E**,E,M,A,P,L,X]

Since , stop, undo last swap [**E**,E,A,M,P,L,X]

Swap the pivot to [A,E,**E**,M,P,L,X]

QuickSort([A,E])

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

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

Parition([**M**,P,L,X])

(P > M)

(X > M) 2 (L M)

Swap 2 and 1 [**M**, L, P, X]

(P > M)

(L < M)

Swap 2 and 1 [**M**, P, L, X]

Undo last swap [**M**, L, P, X]

Swap pivot with : [L, **M**, P, X]

QuickSort([L])

QuickSort([P, X])

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