Data Structures and Algorithms

Dr. L. Rajya Lakshmi

Analysis of insertion sort

for j ← 1 to n-1 do	n
key ← A[j]	n-1
{insert A[j] into the sorted	
sequence A[0. .j-1]}	
i ← j-1	n-1
while i ≥ 0 and A[i] > key do	$\sum_{j=1}^{n-1} t_j$
A[i+1] ← A[i]	$\sum_{j=1}^{n-1} (t_j - 1)$
i - -	$\sum_{j=1}^{n-1} (tj - 1)$
A[i+1] ← key	n-1

Analysis of insertion sort

T(n) = n + (n-1) + (n-1) + $\sum_{j=1}^{n-1} t_j$+ $\sum_{j=1}^{n-1}(tj - 1)$+ $\sum_{j=1}^{n-1}(tj - 1)$+ (n-1)

= n +3(n-1) + (n-1)n/2 + (n-1)(n-2)/2

= an^2 + bn + c

- O-notation describes an upper bound; when we use it to bound the worst-case running time of an algorithm, we have a bound on the running time of that algorithm on every input
- Does $\Theta(n^2)$ bound on the worst-case running time of insertion sort imply $\Theta(n^2)$ bound on every input?

Selection sort

Algorithm Selection_Sort(A[0. .n-1], n)

```
for i ← 0 to n-2 do
    m ← i
    for j ← i+1 to n-1 do
        if A[j] < A[m]
            m ← j
    swap A[i] and A[m]
```

- T(n) = $\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} c$

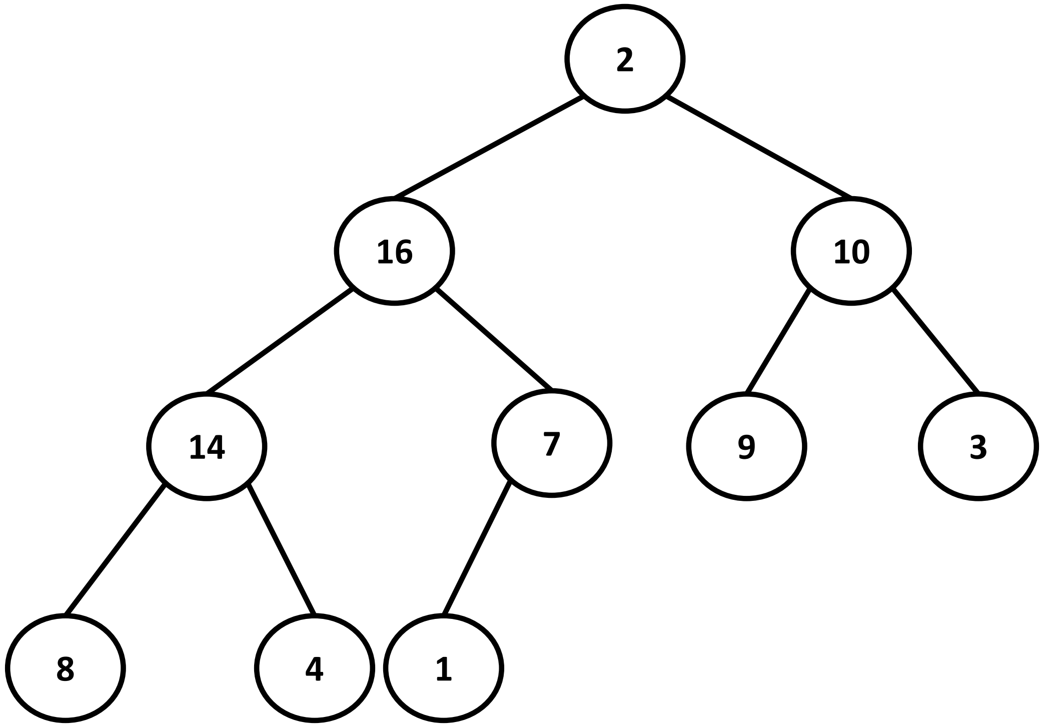
  = $\sum_{i=0}^{n-2} c(n-i-1)$ = c$\sum_{i=0}^{n-2}(n-1)$ - c$\sum_{i=0}^{n-2} i$

  = c(n-1)(n-1) – c(n-2)(n-1)/2 = cn(n-1)/2

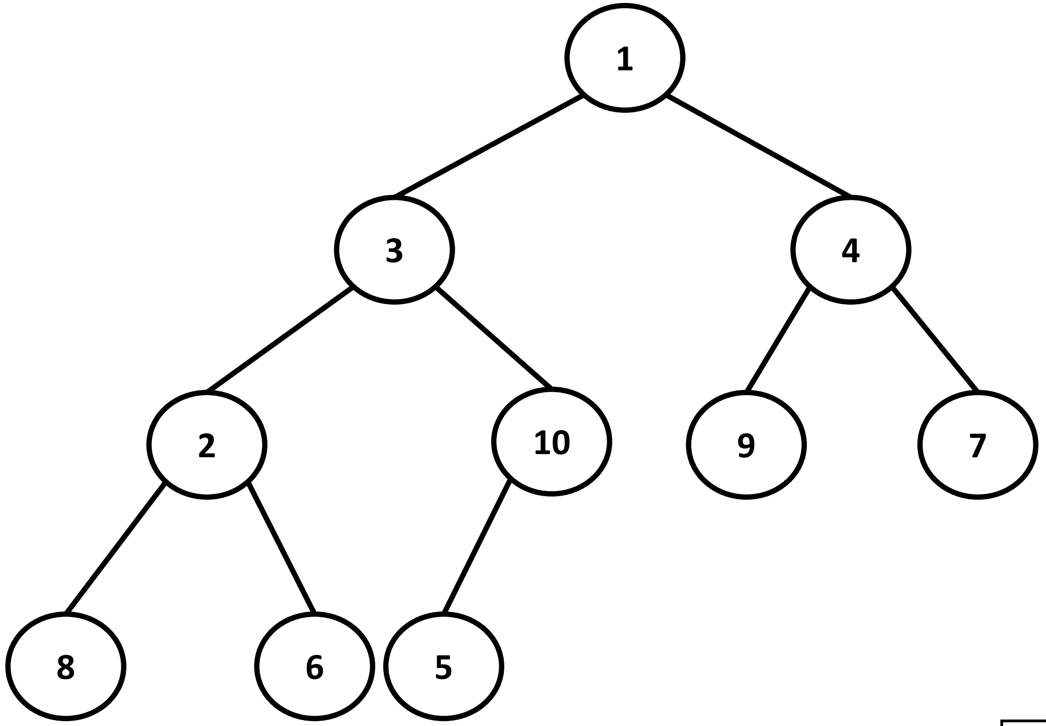
Heapsort

Algorithm Max_Heapify(A, i)

```
l ← Left(i)
r ← Right(i)
if l ≤ A.heap_size and A[l] > A[i]
        largest ← l
else
        largest ← i
if r ≤ A.heap_size and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i], A[largest]
        Max_Heapify(A, largest)
```

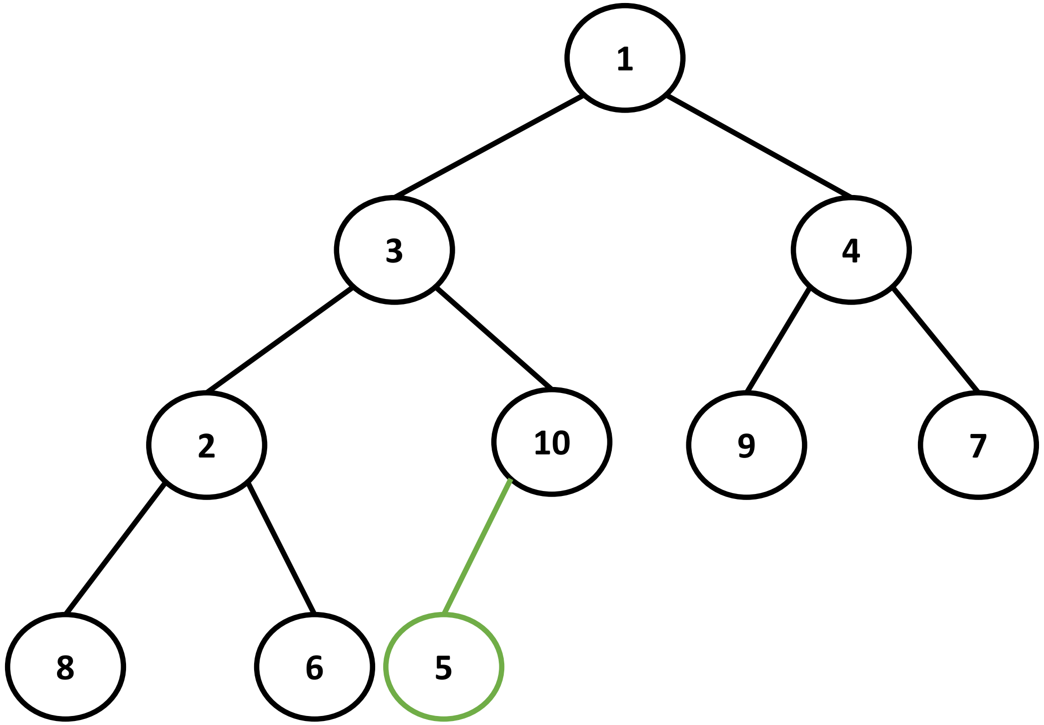


Heapsort

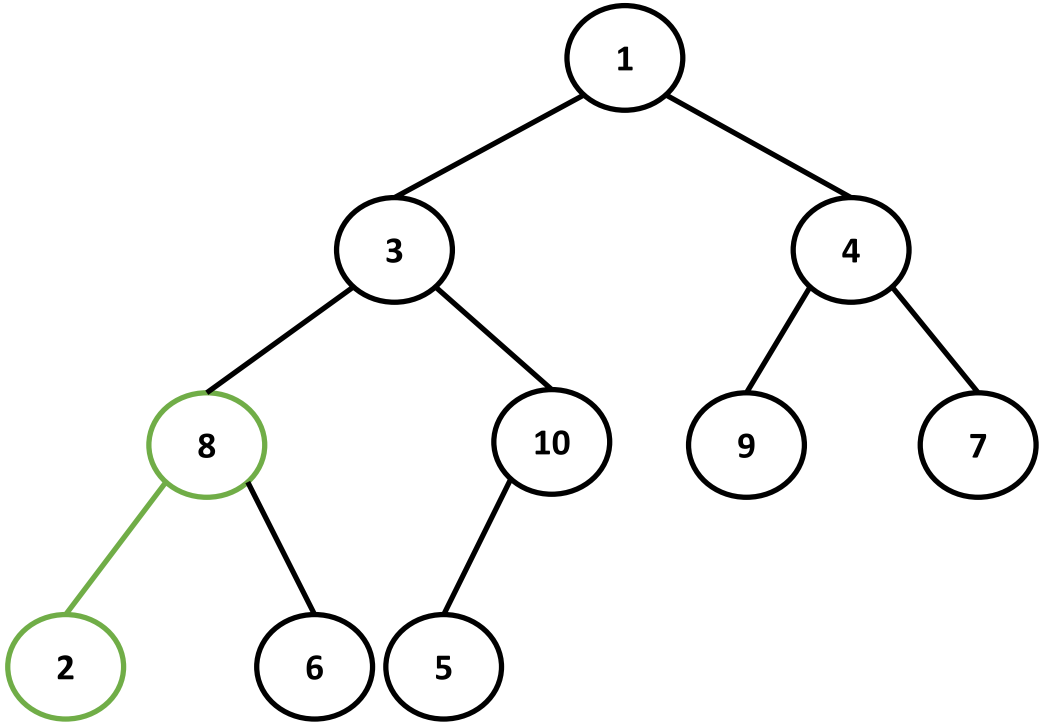


1	3	4	2	10	9	7	8	6	5

Heapsort

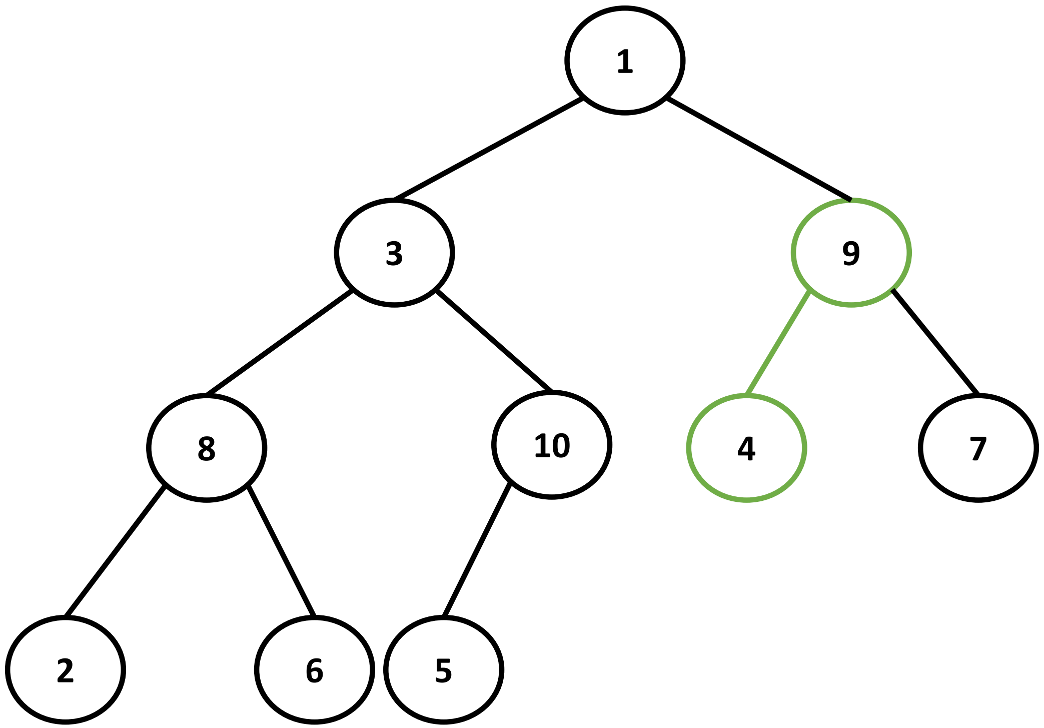


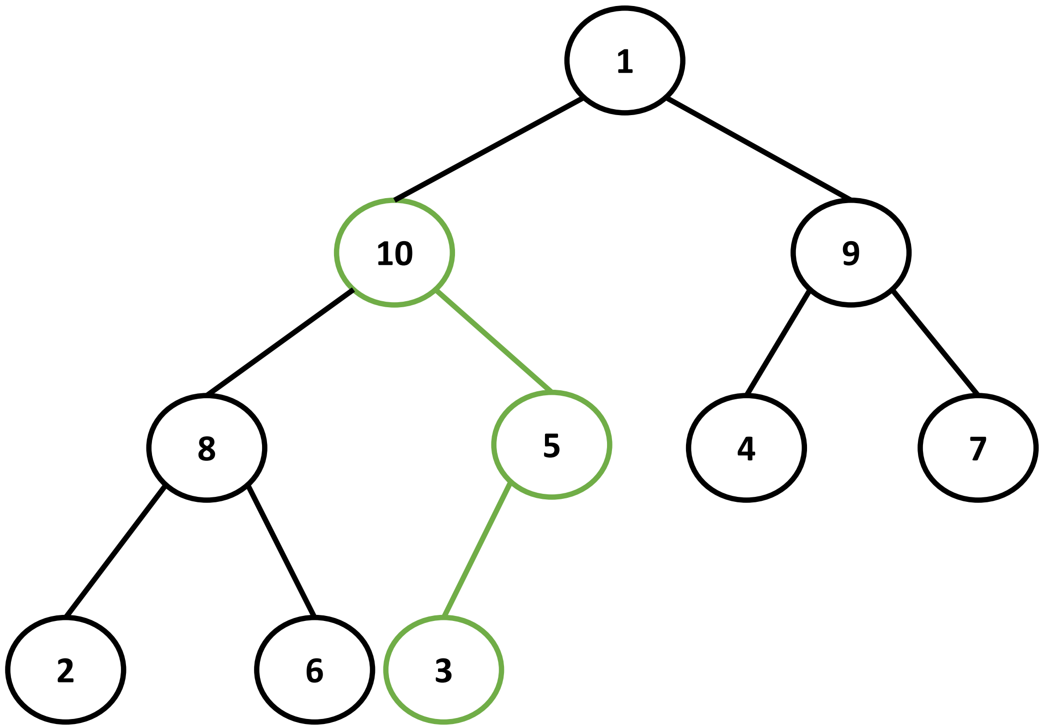
1	3	4	2	10	9	7	8	6	5



1	3	4	8	10	9	7	2	6	5

Heapsort

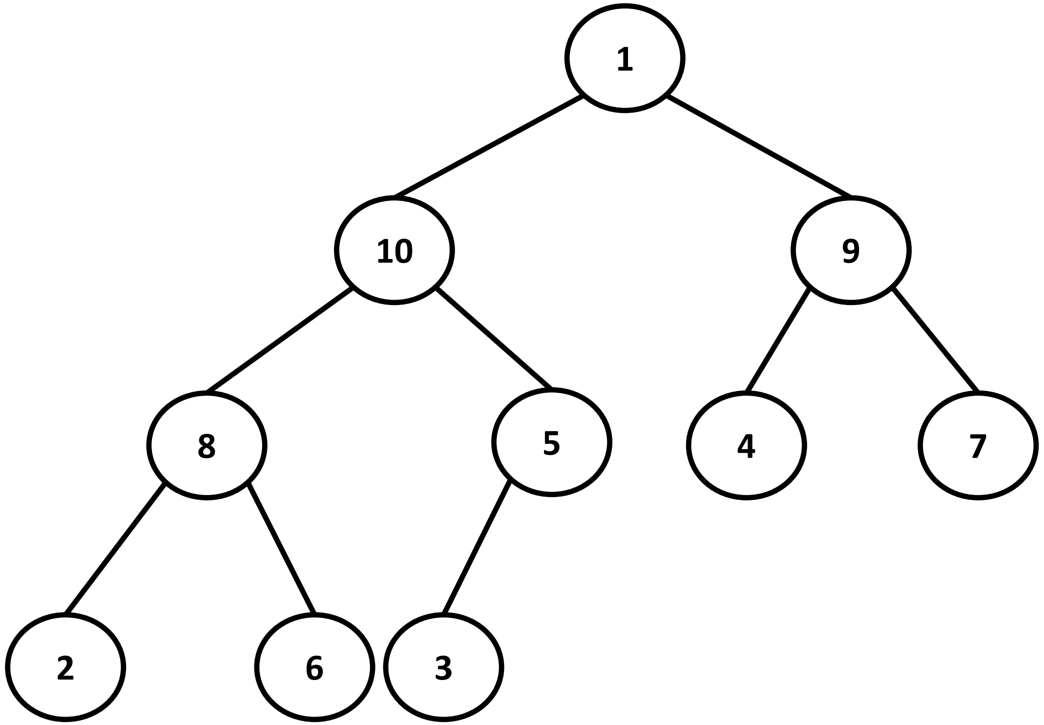




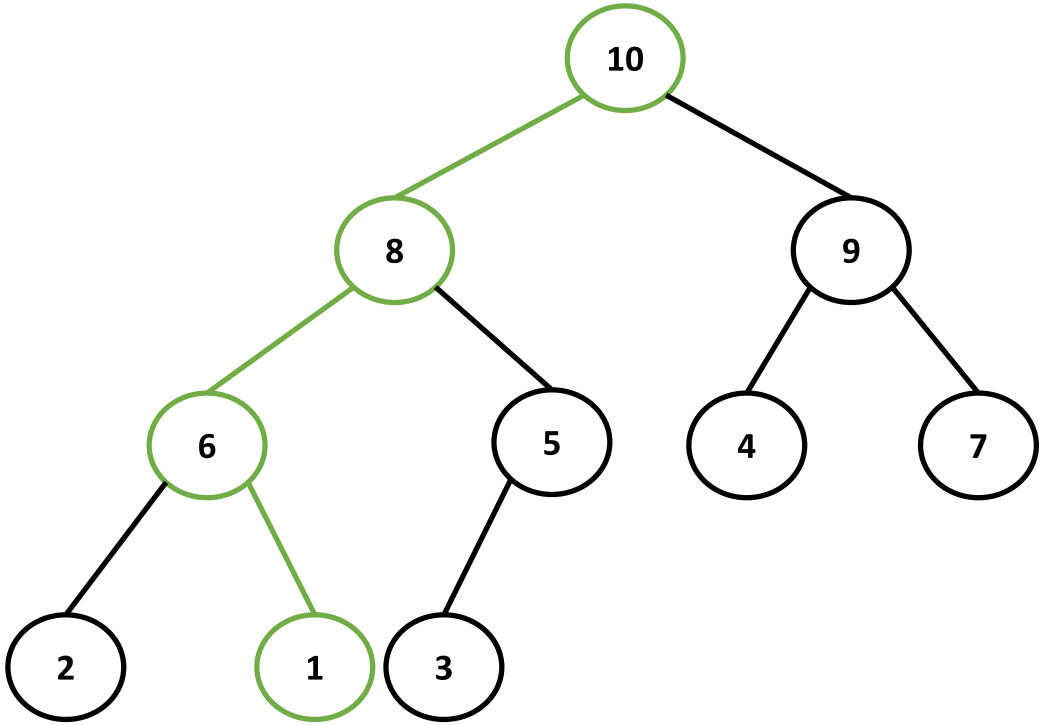
1	3	9	8	10	4	7	2	6	5

1	10	9	8	5	4	7	2	6	3

Heapsort



1	10	9	8	5	4	7	2	6	3



10	8	9	6	5	4	7	2	1	3

Heapsort

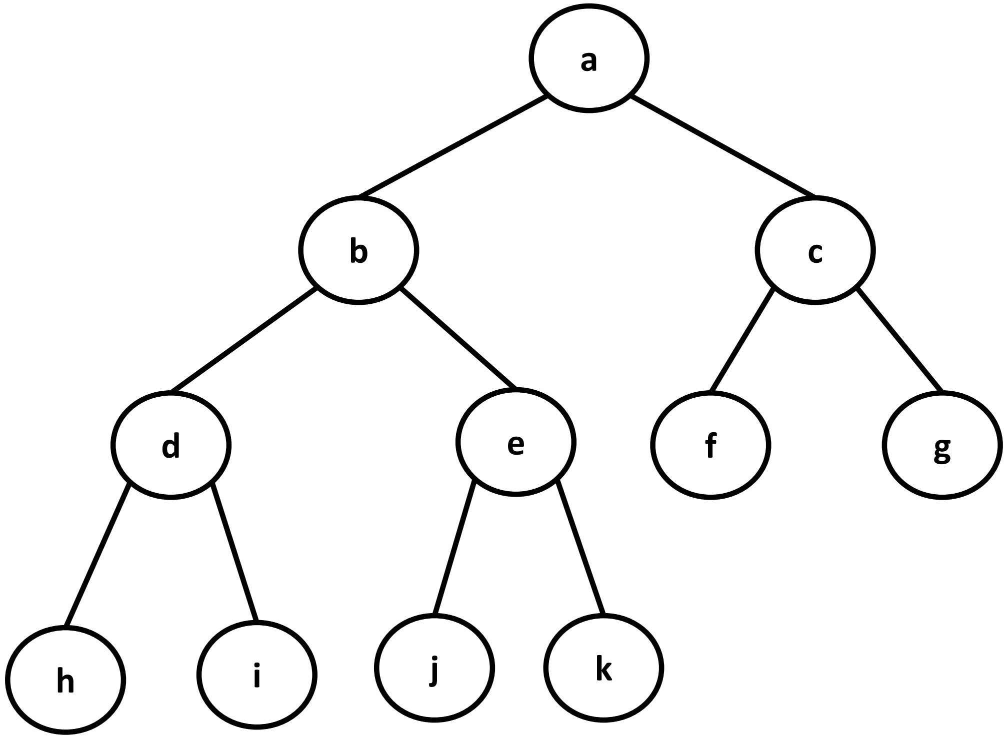
Algorithm Build_Max_Heap(A)

    A.heap_size ←A.length

    for i ← $\lfloor A.lenght/2 \rfloor$ downto 1

        Max_Heapify(A, i)

Heapsort



Heapsort

- Time to run Max_Heapify at a node varies with the height of the node
- Time required to run Max_Heapify at a node of height h is O(h)
- An n element heap has height $\lfloor \log(n) \rfloor$
- At height "h", there would be at most $\lceil n/2^{h+1} \rceil$ nodes
- The total cost of Build_Max_Heap:

$$\sum_{h=1}^{\lfloor \log(n) \rfloor} \lceil n/2^{h+1} \rceil \, O(h) \text{ which is } O(n \sum_{h=1}^{\lfloor \log(n) \rfloor} \lceil h/2^{h} \rceil$$

Heapsort

$\sum_{i=0}^{\infty} x^i = \frac{1}{(1-x)}$ if |x| < 1 {differentiate both sides}

$\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$ {multiply both sides by x}

$\sum_{i=1}^{\infty} i x^{i} = \frac{x}{(1-x)^2}$

Taking x = ½ yields,

$\sum_{i=1}^{\infty} i x^{i} = \frac{1/2}{(1-1/2)^2}$ = 2

O(n $\sum_{h=1}^{\lfloor \log(n) \rfloor} \lceil h/2^h \rceil$ is O(n $\sum_{h=1}^{\infty} \lceil h/2^h \rceil$ which can be written as O(n $\frac{1/2}{\left(1-\frac{1}{2}\right)^2}$) which is O(n)

Heapsort

- Consider an array A[1 . . n], n is A.length
- Run Build_Max_Heap on A
- The largest element is sitting in A[1]
- Swap A[1] and A[n], decrement A.heap_size by 1, and run Max_Heapify on the new root
- Repeat the above step until A.heap_size becomes 1

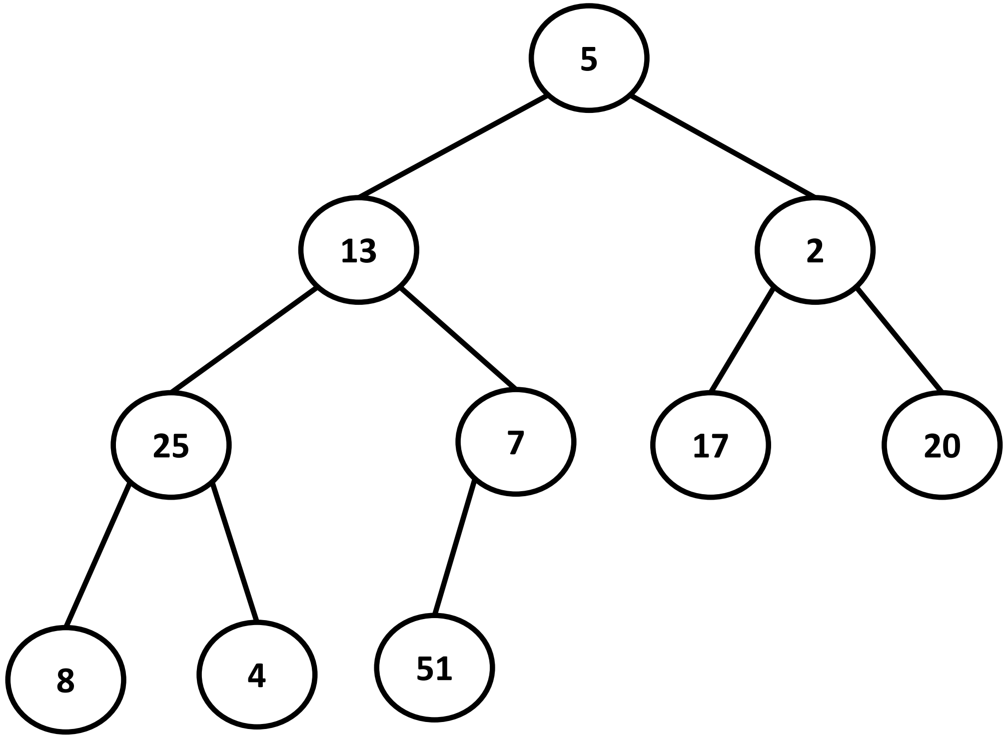
Heapsort

Algorithm Heapsort(A)

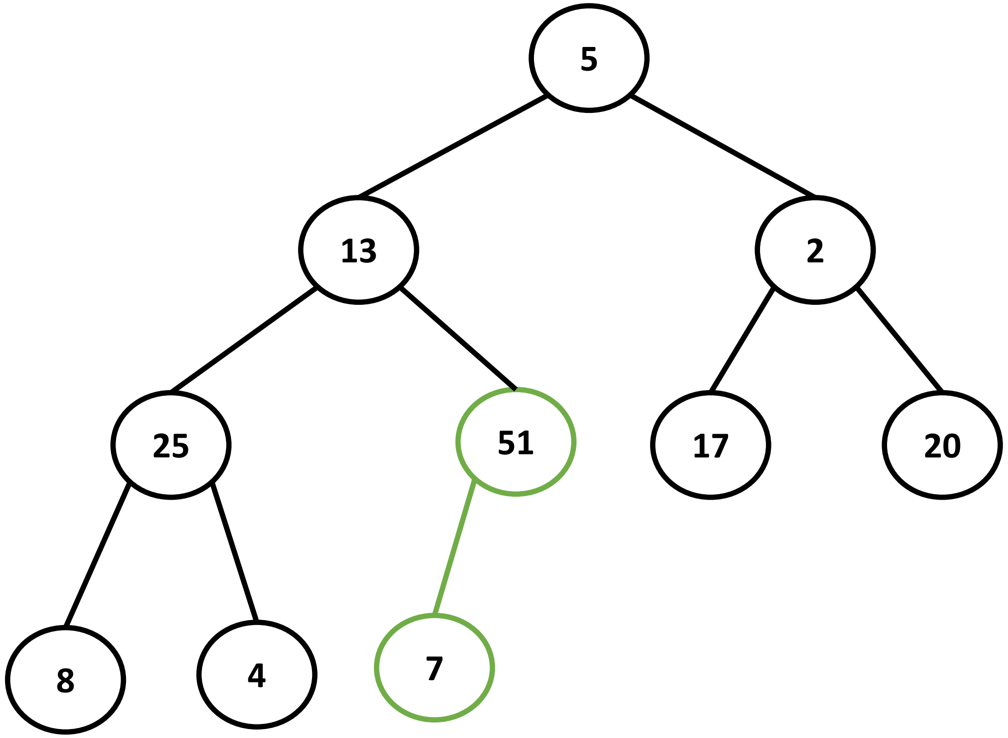
```
        Build_Max_Heap(A)
        for i ← A.length downto 2
                swap A[1] and A[i]
                Max_Hepify(A, 1)
```

- Build_Max_Heap() takes O(n) time
- Each of n-1 calls of Max_Heapify() takes O(log n) time
- Running time is O(n log n)

Heapsort

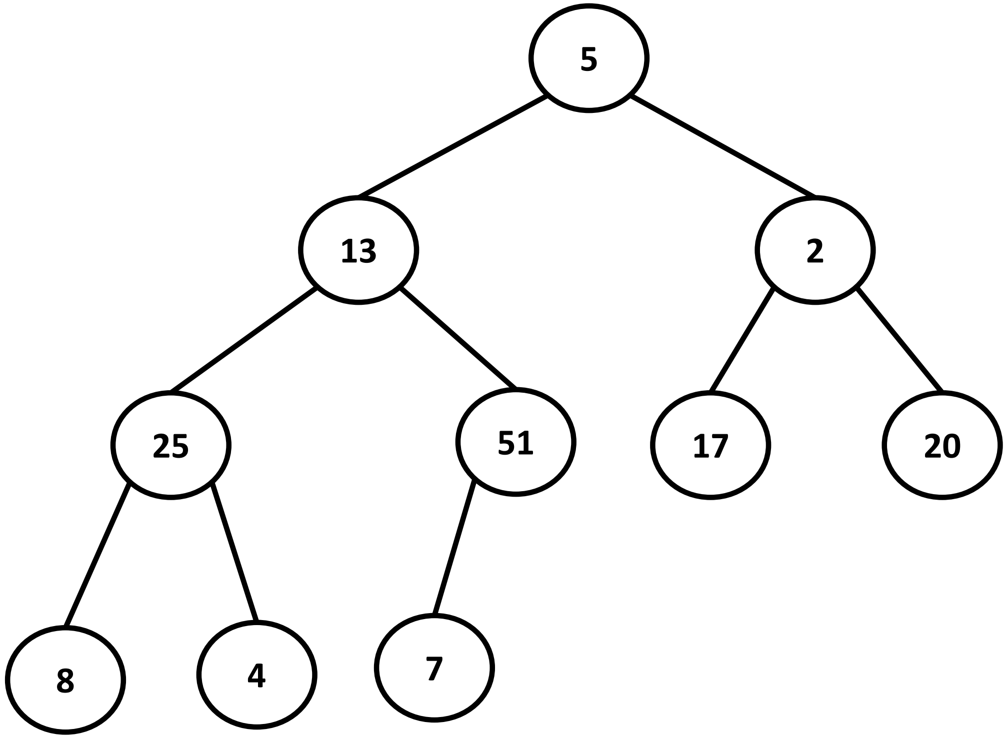


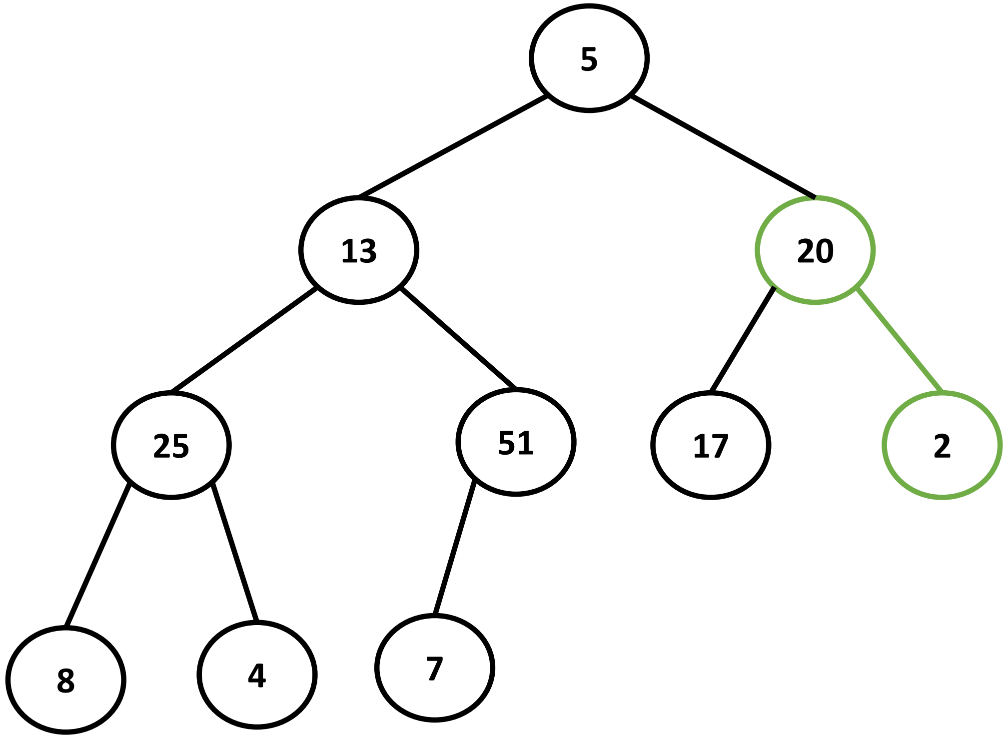
5	13	2	25	7	17	20	8	4	51



5	13	2	25	51	17	20	8	4	7

Heapsort

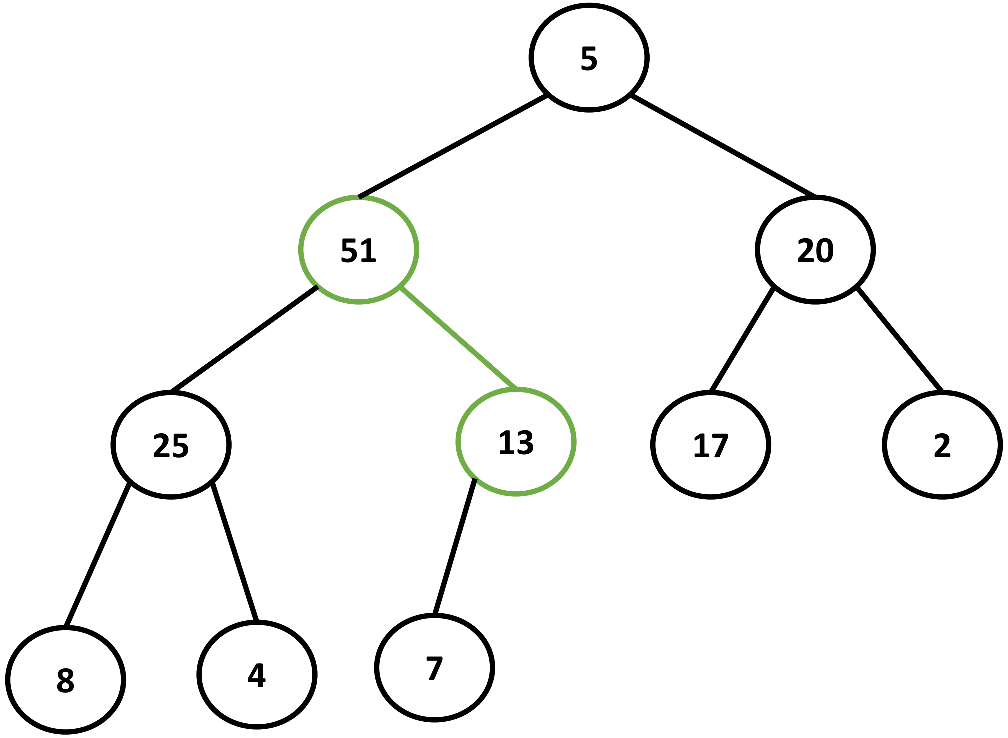




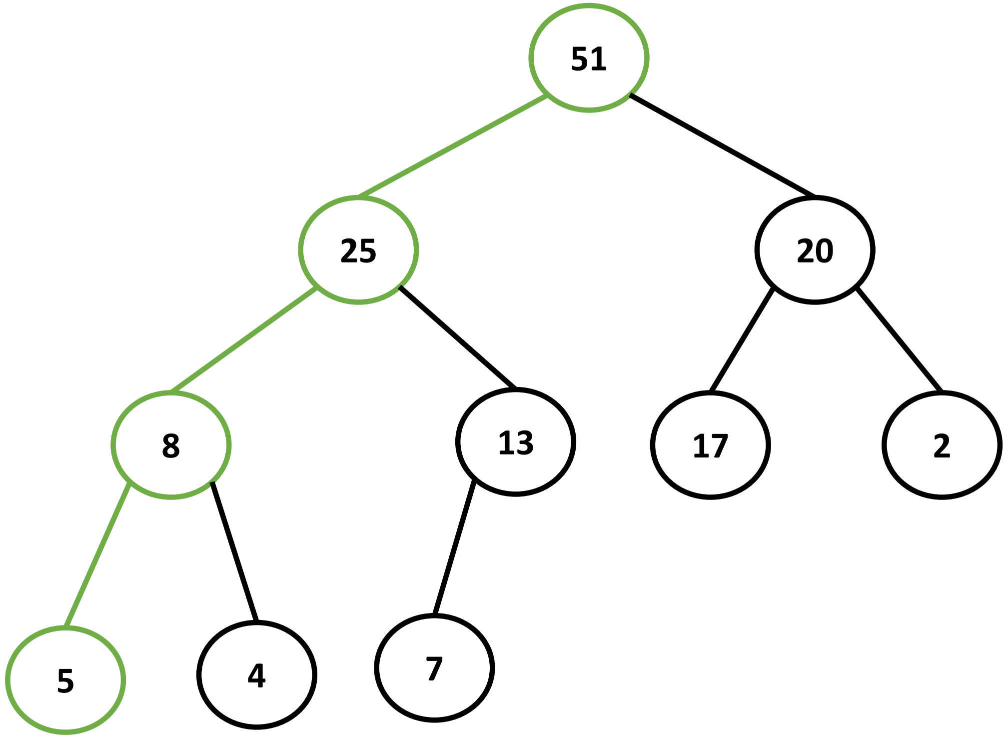
5	13	2	25	51	17	20	8	4	7

5	13	20	25	51	17	2	8	4	7

Heapsort

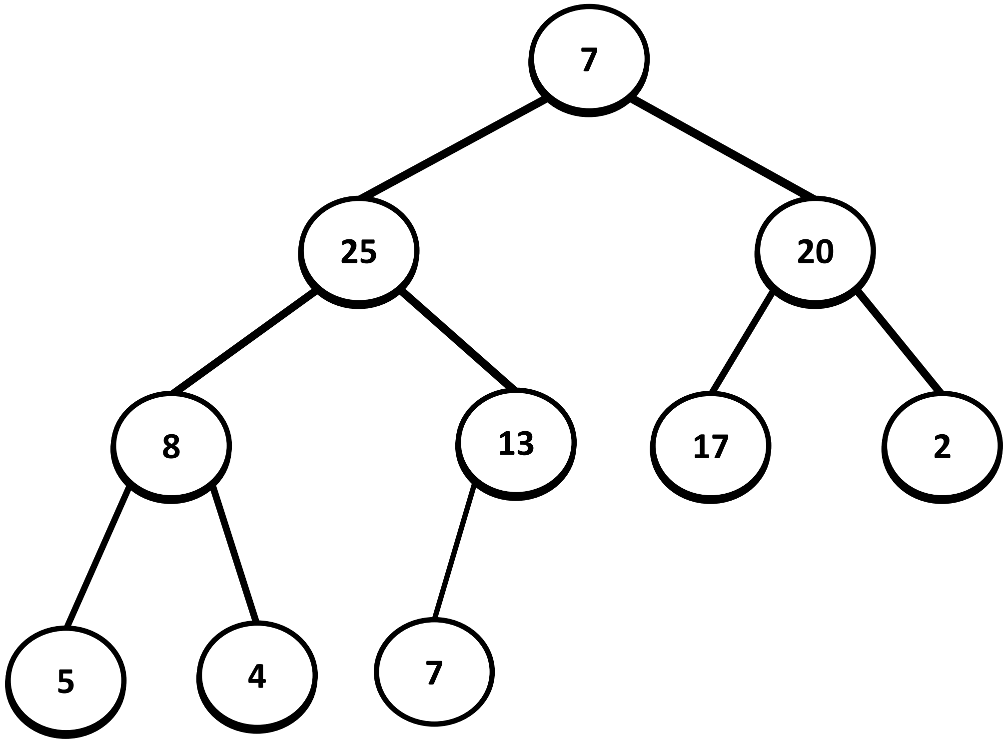


5	51	20	25	13	17	2	8	4	7



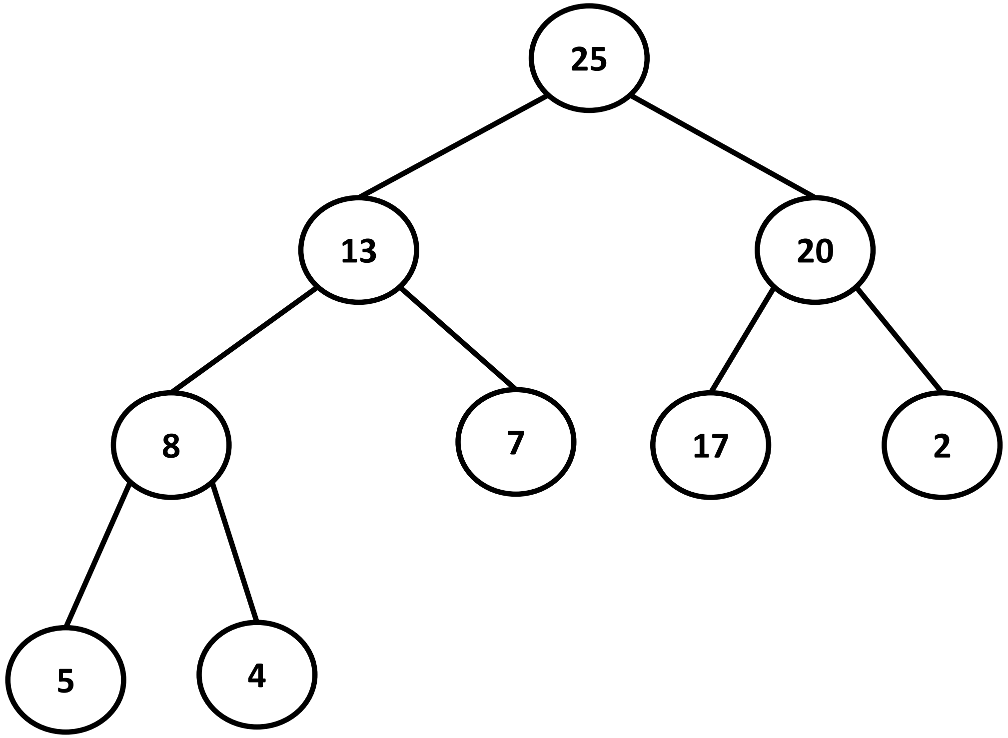
51	25	20	8	13	17	2	5	4	7

Heapsort



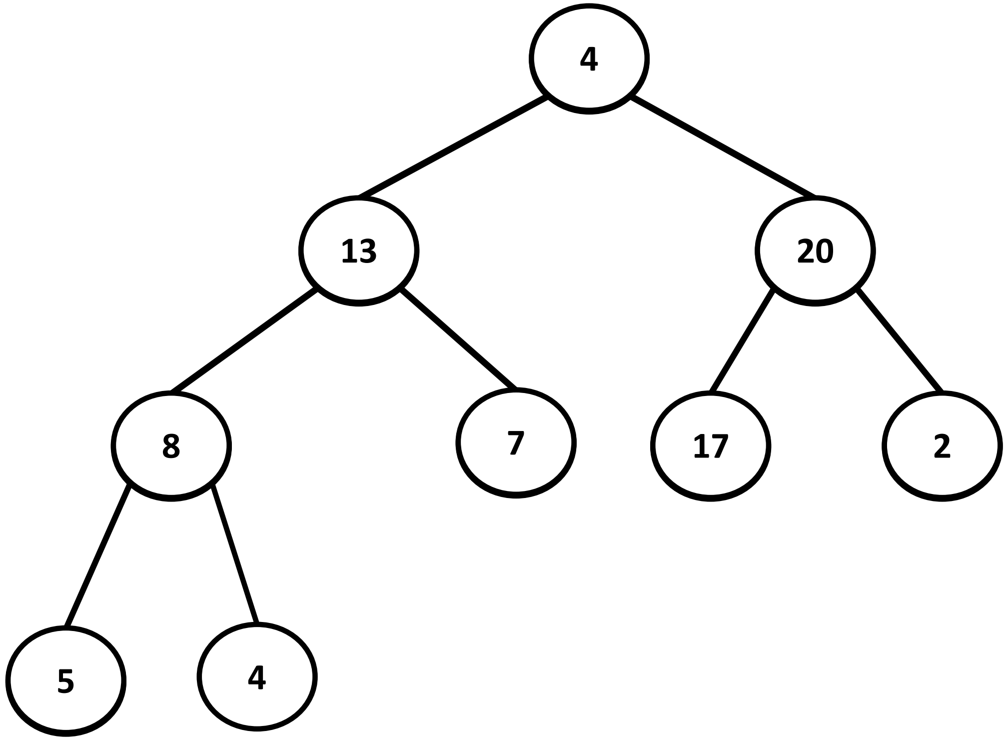
51	25	20	8	13	17	2	5	4	7

7	25	20	8	13	17	2	5	4	51



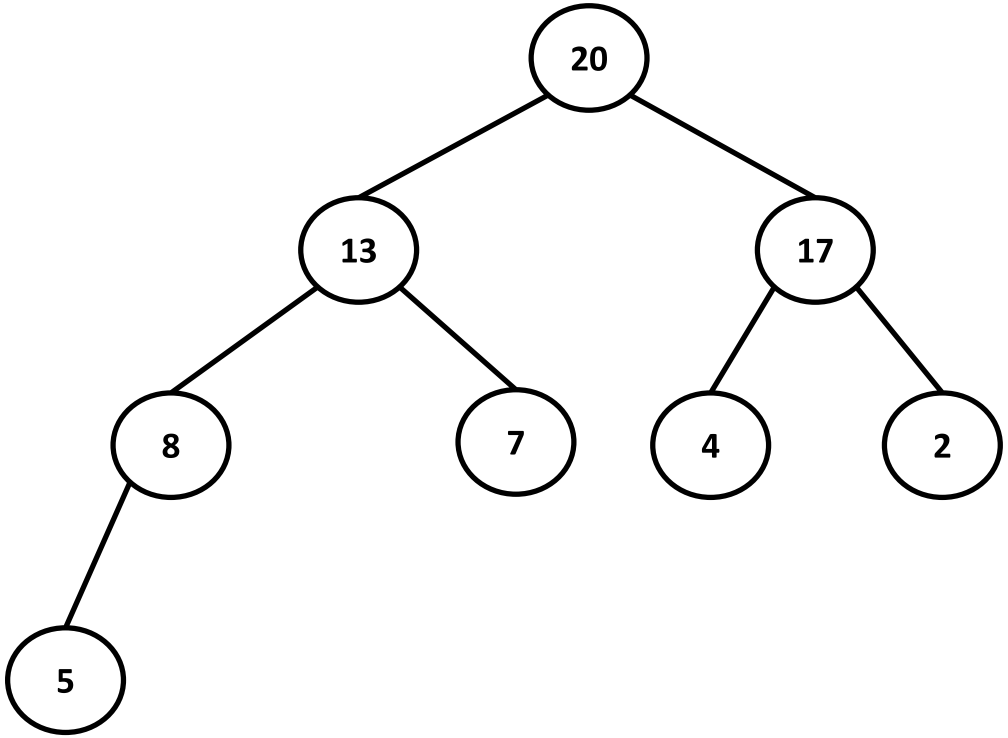
25	13	20	8	7	17	2	5	4	51

Heapsort



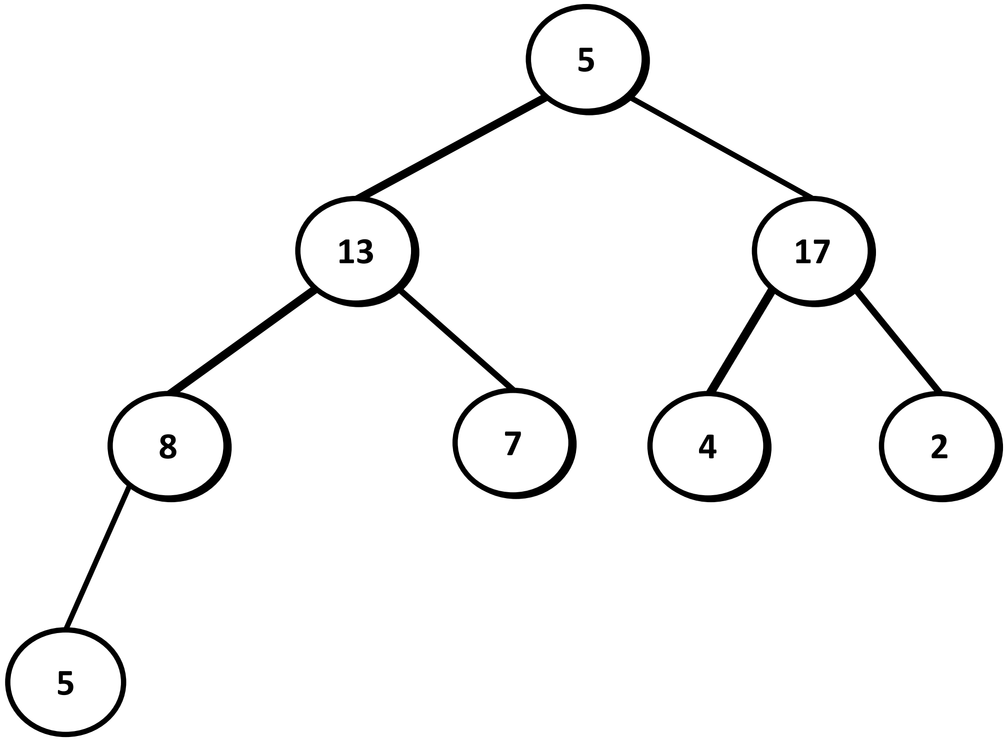
25	13	20	8	7	17	2	5	4	51

4	13	20	8	7	17	2	5	25	51



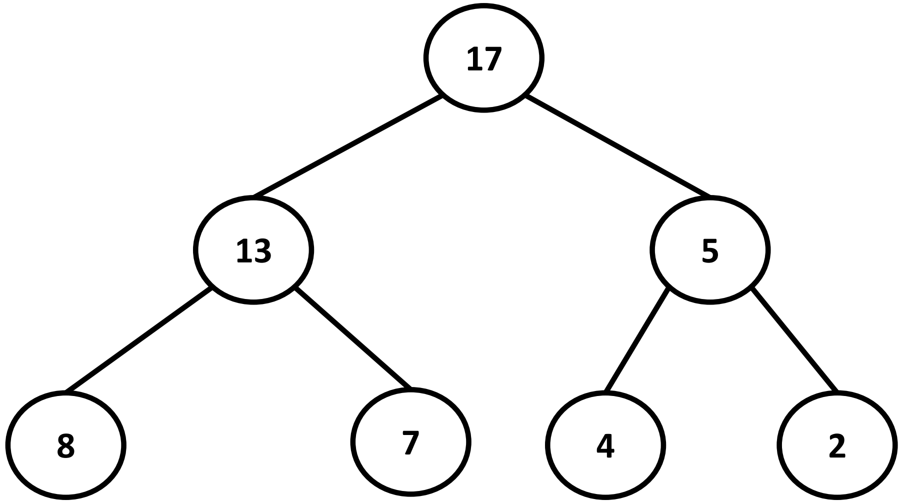
20	13	17	8	7	4	2	5	25	51

Heapsort



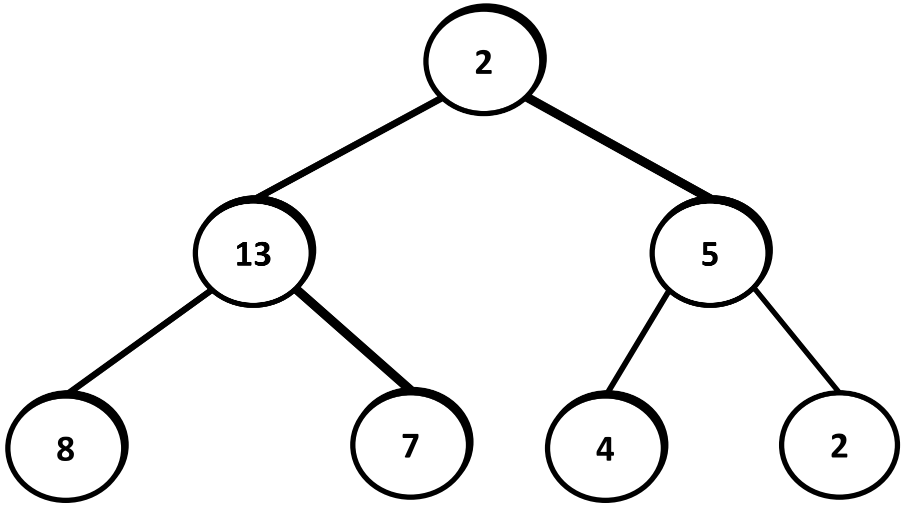
20	13	17	8	7	4	2	5	25	51

5	13	17	8	7	4	2	20	25	51



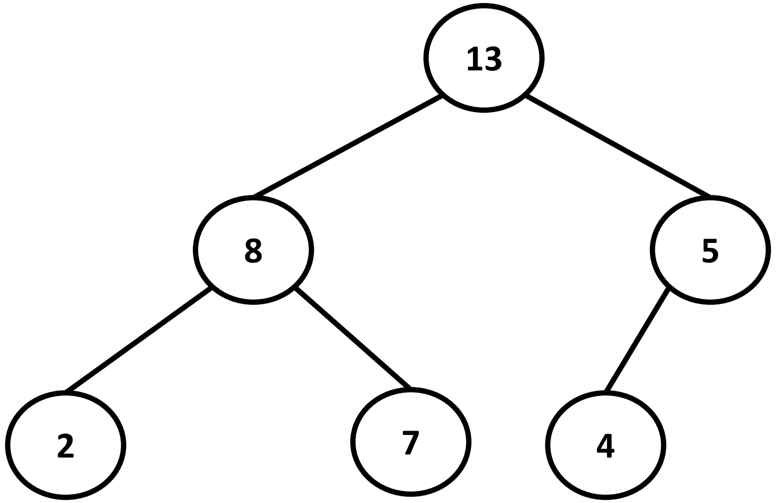
17	13	5	8	7	4	2	20	25	51

Heapsort



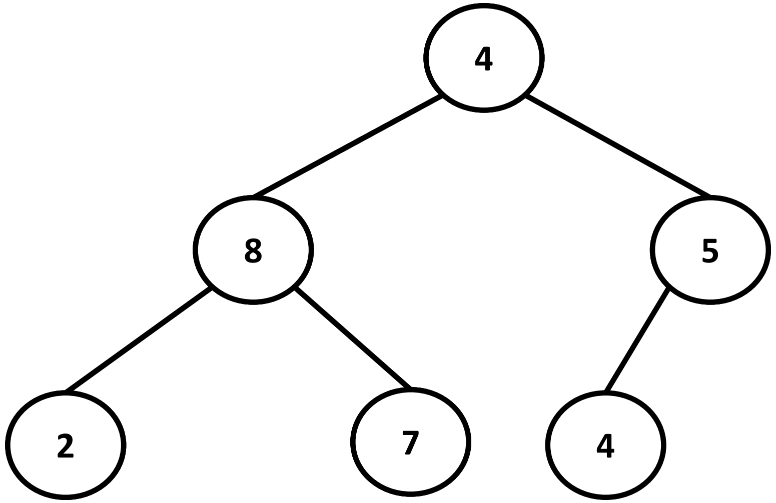
17	13	5	8	7	4	2	20	25	51

2	13	5	8	7	4	17	20	25	51



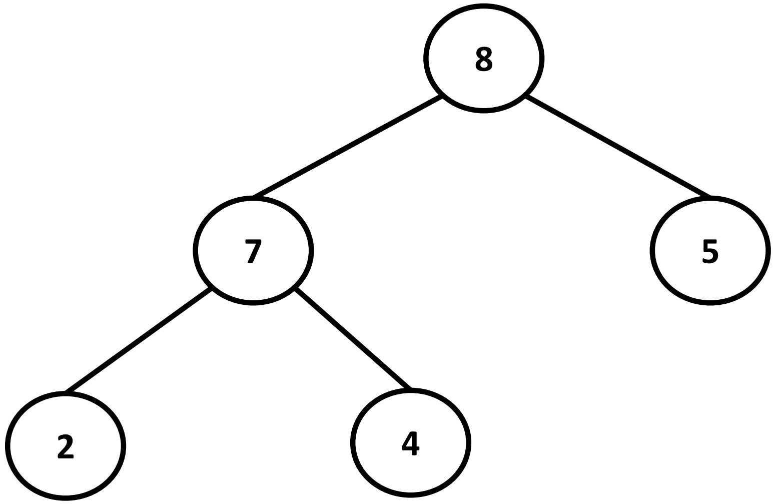
13	8	5	2	7	4	17	20	25	51

Heapsort



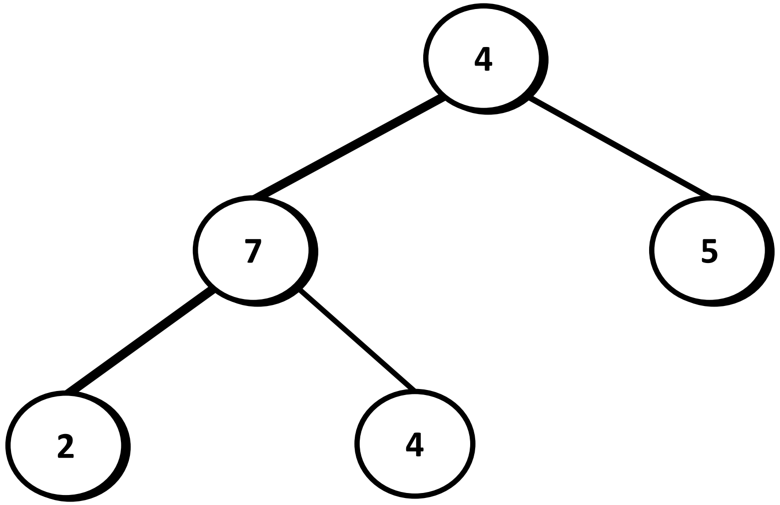
13	8	5	2	7	4	17	20	25	51

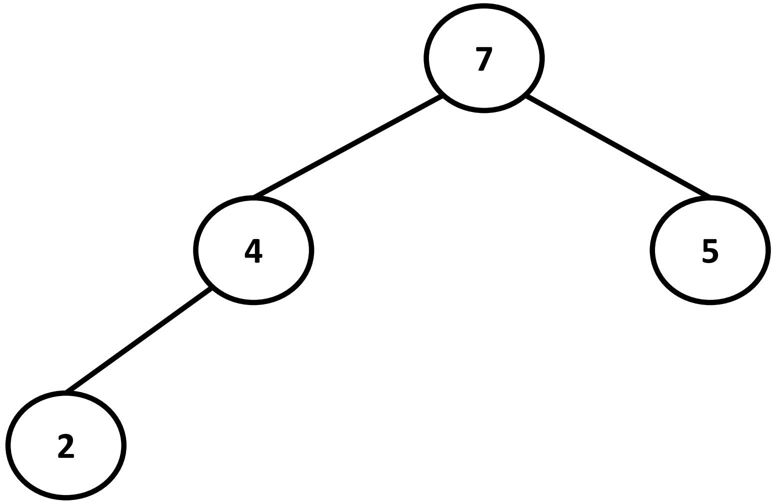
4	8	5	2	7	13	17	20	25	51



8	7	5	2	4	13	17	20	25	51

Heapsort



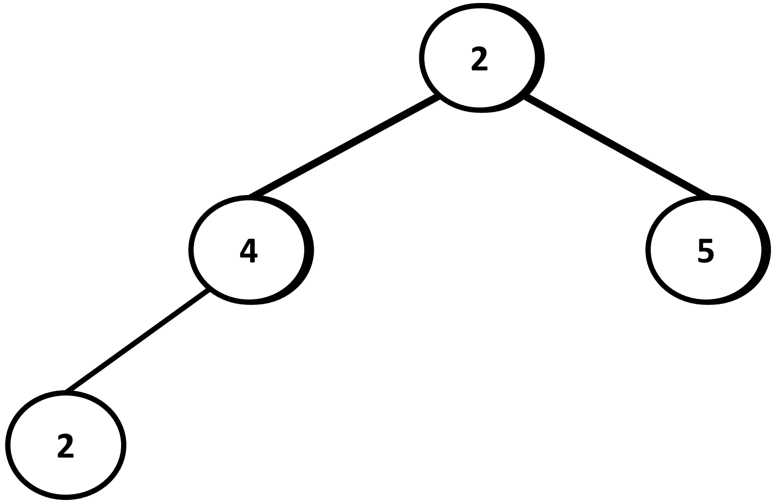


8	7	5	2	4	13	17	20	25	51

7	4	5	2	8	13	17	20	25	51

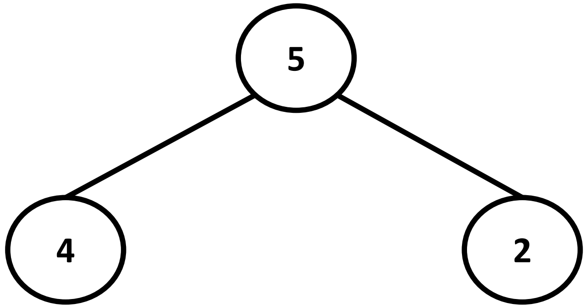
4	7	5	2	8	13	17	20	25	51

Heapsort



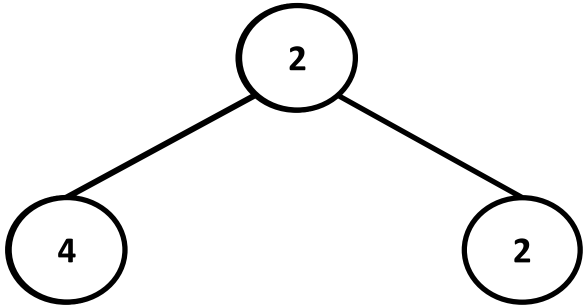
7	4	5	2	8	13	17	20	25	51

2	4	5	7	8	13	17	20	25	51



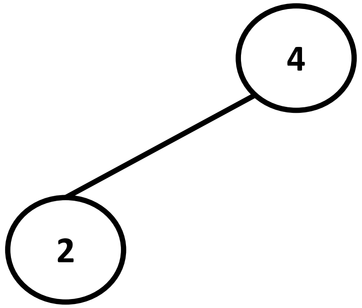
5	4	2	7	8	13	17	20	25	51

Heapsort



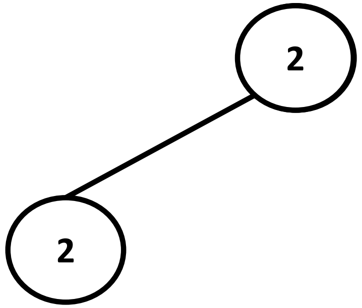
5	4	2	7	8	13	17	20	25	51

2	4	5	7	8	13	17	20	25	51



4	2	5	7	8	13	17	20	25	51

Heapsort



2	4	5	7	8	13	17	20	25	51

2	4	5	7	8	13	17	20	25	51