(1)

No:

Date: / /

CO 322

Data Structures and Algorithms

Assignment 1 - part 1

E/16/388

Weerasundara W.M.T.M.P.B.

1.1 increasing order of big-O growth,

$$4 \lg(\lg n) < 4 \lg n < 5n < n^{\frac{1}{2}} (\lg n)^4 < \left(\frac{n}{4}\right)^{\frac{n}{4}} < 5^{5n}$$

1.2 Master theorem,

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non negative integers by the recurrence,

$$T(n) = a\,T(n/b) + f(n)$$

where n/b is either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$, Then $T(n)$ has following assymptotic bounds.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficient-ly large n, then $T(n) = \Theta(f(n))$

In each 3 cases, we compare $f(n)$ with $n^{\log_b a}$. Larger of 2 functions determine the solution to recurrence.

1.3.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$,
then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$
then $T(n) = \Theta(n^{\log_b a} \lg n)$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if
$a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficient
-ly large n,
then $T(n) = \Theta(f(n))$

In each 3 cases, we compare $f(n)$ with $n^{\log_b a}$. Larger
of 2 functions determine the solution to recurrence.

1.3.

* $T(n) = 4T(n/4) + 5n$

from $T(n) = a\,T(n/b) + f(n)$,
$a = b = 4$ and $f(n) = 5n$

$n^{\log_b a} = n^{\log_4 4} = n$

$$\lim_{n \to \infty} \frac{f(n)}{n^{\log_4 4}} = 5$$

$\therefore f(n) = \Theta(n^{\log_b a})$

$\therefore T(n) = \Theta(n \lg n)$ //

* $T(n) = 5\,T(n/4) + 4n$

from $T(n) = a\,T(n/b) + f(n)$,

$a = 5$, $b = 5$, $f(n) = 4n$

$$n^{\log_4 5} = n^{1.16}$$

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_4 5}} = \lim_{n\to\infty} \frac{4n}{n^{1.16}} \to 0$$

$\therefore f(n) = O(n^{\log_b a - \varepsilon})$

$\therefore T(n) = \Theta(n^{\log_4 5})$

$= \Theta(n^{1.16})$ //

* $T(n) = 25\,T(n/5) + n^2$

from $T(n) = a\,T(n/b) + f(n)$,

$a = 25$, $b = 5$, $f(n) = n^2$

$$n^{\log_5 25} = n^2$$

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_5 25}} = \lim_{n\to\infty} \frac{n^2}{n^2} = 1$$

$\therefore f(n) = \Theta(n^{\log_b a})$

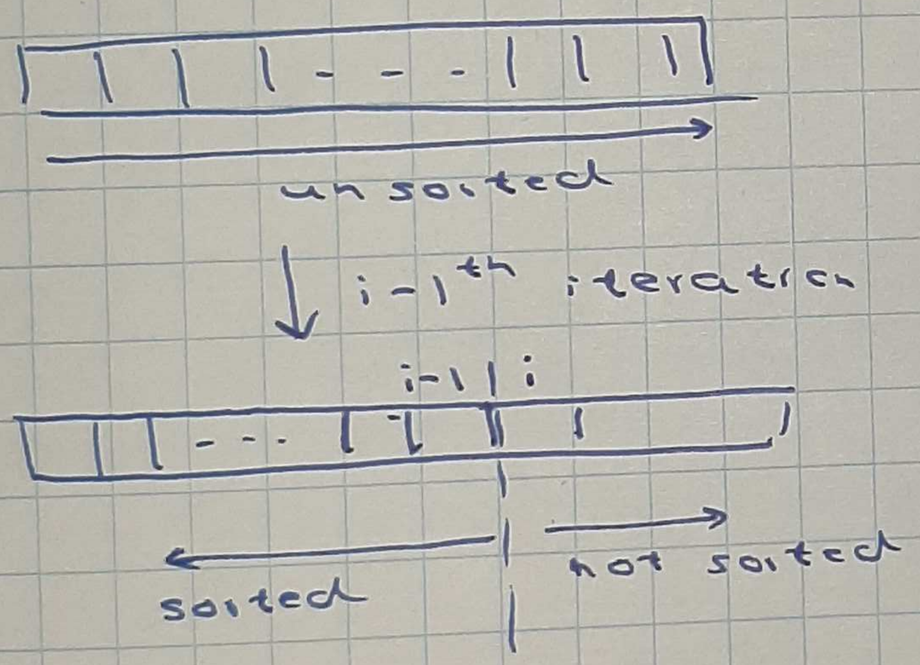
$\therefore T(n) = \Theta(n^{\log_b a} \log n)$

$= \Theta(n^2 \log n)$ //

1.4

(a) In the given bubble sort algorithm, during each iteration of the outer loop, smallest elements in 'a' will be moved to the ~~rt.~~ left side of 'a'. At each iteration up to position current i, elements will be sorted.

for each iteration of outer loop, inner loop will select the current last element in 'a' and compare it with the previous element and swap if last element is small. Inner loop will iterate from end to $i+1^{th}$ position while doing the comparison with $j-1^{th}$ element. Therefor at the end of $i-1$ iterations of outer loop, $i-1$ number of elements are sorted and stored in left side and elements that are larger or equal than $i-1^{th}$ element are stored in right side of 'a'. Elements in right of $i-1$ could be sorted in some scenarios as well.



~~A[i+1] ← A[i]~~

(b) For worst case in buble sort, 'a' must be reverse sorted. Therefor at each step of inner loop, we would have to perform a swap.

```
for (i = 1 ; i <= n-1 ; i++)        — (1)   ——————→ n-1
  {
      for (j = n ; j >= i+1 ; j--)  — (2)   ——————→ n
          if (a[j].key < a[j-1].key)  — (3)
              swap the records a[j] and a[j-1] — (4)
  }
```

(b) for worst case in buble sort, (a) must be reverse sorted. Therefor at each step of inner loop, we would have to perform a swap.

```
for (i = 1 ; i <= n-1 ; i++)        — ①        ⟶ n-1
    {
        for (j = n ; j >= i+1 ; j--)    — ②    ⟶ ∝
            if ( a[j].key < a[j-1].key )    — ③
                swap the records a[j] and a[j-1]    — ④
    }
```

- loop represented by ① will ~~exec~~ iterate $n-1$ times
- loop represented by ② will iterate $\sum_{i=i+1}^{n}$ times ; $i = 1, \dots n-1$
- due to worse case scenario, both ③ and ④ will run $\sum_{i=i+1}^{n}$ times ; $i = 1, 2, \dots n-1$

~~each iteration of~~ Inner loop will have $n-i$ iterations per each iteration of outer loop.

$$T(n) = \sum_{i=1}^{n-1} (n-i)\,c$$

$$= c\,[\,n-1 + n-2 + \cdots + (n-(n-2)) + (n-(n-1))\,]$$

$$= c\,[\,n-1 + n-2 + \cdots + 2 + 1\,]$$

$$= c\,\frac{(n-1)}{2}\,(1 + n-1)$$

$$T(n) = O(n^2) \;//$$

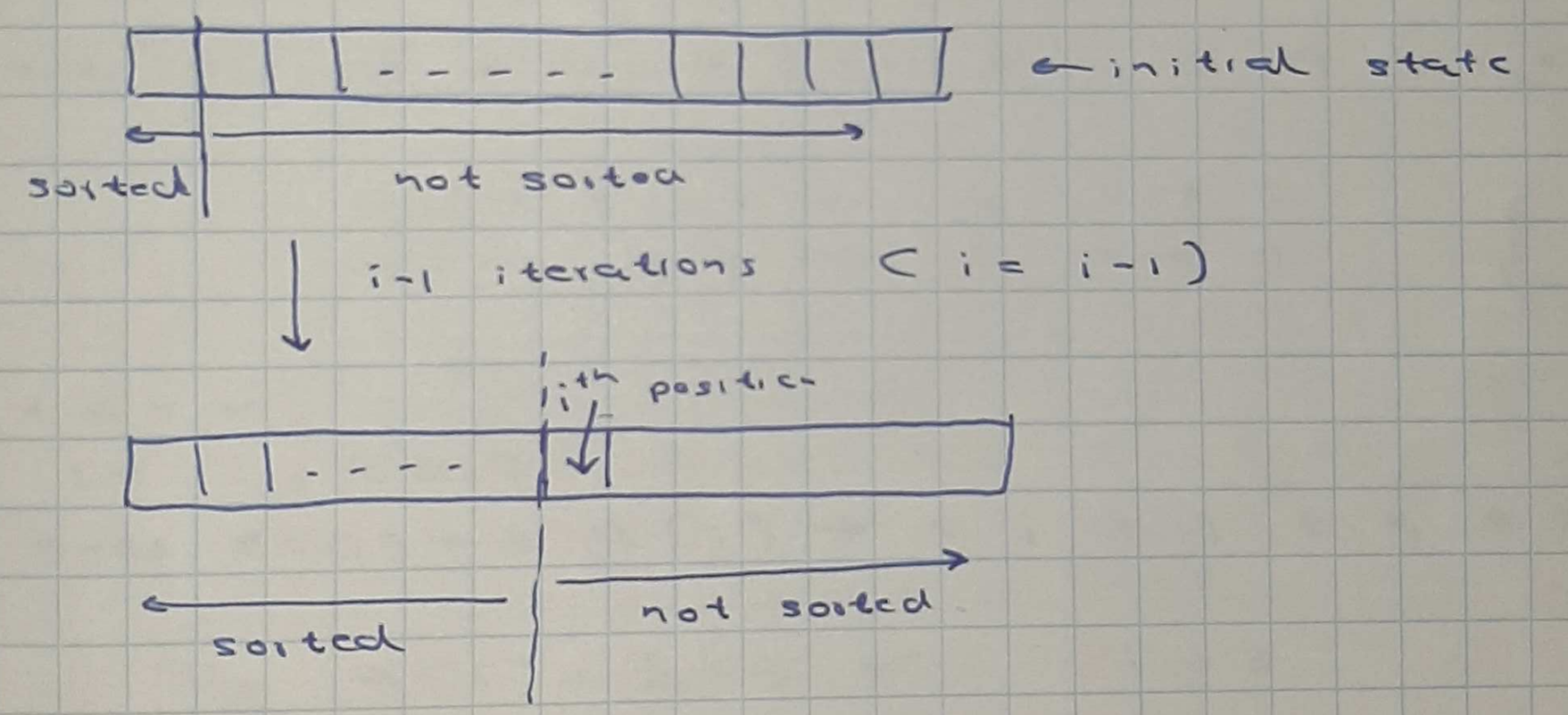
∴ worst case complexity for bubble sort is in $O(n^2)$ //

1.5 insertion sort

(c) In the given insertion sort algorithm, at each iteration of the inner while loop, most left element in the unsorted part ~~of~~ of 'a' is selected and store it in the correct position of the sorted part of 'a'. Inner while loop exits when suitable position is found for ~~a~~ currently considering element. ~~for i iterations.~~

iteration	i	number of sorted elements
1	2	2
2	3	3
3	:	:
:	:	:
i-2	i-1	i-1
i-1	i	i

for at the end of $i = i-1^{th}$ iteration, i-1 numbers will be sorted and stored in the left side of i^{th} position of 'a'.



(d)
```
for (i=2 ; i<=n ; i++)          ---> n-2 -> c1
  {
    j = i ;                     ---> c2
    while ( key of a[j] < key of a[j-1]) -> i-1 -> c3
      {
        swap records a[j] and a[j-1] ; -> c4
        j = j-1 ;               --- c5
      }
  }
```

- for worst case, input set of values are reverse sorted.
- for worst case, while loop should ~~execute~~ iterate i-1 times per each iteration of outer loop

No :

Date: __/__/____

$$T(n) = \sum_{i=2}^{n} (c_2 + (i-1)(\underbrace{c_3+c_4+c_5}_{c'}))$$

$$= (n-1)c_2 + c' \sum_{i=2}^{n} (i-1)$$

$$= c_2(n-1) + c'(1+2+\cdots n-1)$$

$$= c_2(n-1) + c'\left(\frac{n-1+1)(n-1)}{2}\right)$$

∴ worst case complexity for insertion sort is in $O(n^2)$ //

~~L6 Heap sort~~

1.7 Quick sort

a. In qurck sort, input array is partitioned to 2 parts. To do thrs, an element is sdected as the pivot and array is splited depending on which elements are larger and which elements are smallei than the pivot. 'v' is the pivot element we choose.

b [3] 1 4 1 5 9 2 6 5 3 ⇒ (A)

- lets consider A(1) as pivot. ∴ v is the largest of 1st 2 distinkt keys.

- Now 'a' will be partitioned by v = 3
  ↳ partition (1, 10, 3) → partition function call

initially l = 1 r = 10

a = 3 1 4 1 5 9 2 6 5 3

swap a(1) ↔ a[10] → 3 1 4 1 5 9 2 6 5 3 → a

a(l) = 3 < 3 ✗ ∴ l → 1
a[r] = 3 ≥ 3 ✓ ∴ r → 7

l ≤ r ✓
l = 1 , r = 7
swap a(1) ↔ a[7] → 2 1 4 1 5 9 3 6 5 3 → a

a(l) = 2 < 3 ✓ ∴ l → 3
a[r] ≥ 3 ✓ ∴ r → 4

l ≤ r ✓
l = 3, r = 4
swap a[3] ↔ a[4] → 2 1 1 4 5 9 3 6 5 3 → a

a(l) = 1 < 3 ∴ l → 4
a[r] = 4 ≥ 3 ∴ r → 3

l ≤ r ✗

return l = 4

2 1 1 | 4 5 9 3 6 5 3

← less than 3 | → larger than or equal to 3

Now, algorithm partition 'a' from 0 to k-1 and k to j.

k = 4

b = 2 1 1 ⟶ quicksort (0, 4-1)

c = 4 5 9 3 6 5 3 ⟶ quicksort (4, 10)

* consider quicksort (1, 3) on 2 1 1

since a(0) > a(2) pivot = 2 , l = 1 , r = 3

swap a(1) ⟷ a(3) → 1 1 2 → a

a(1) = 1 < 2 ∴ l → 3

a(3) = 2 ≥ 2 ∴ r → 2

l ≤ r x

∴ return l = 3 , k = 3

1 1 | 2

less than 2 | larger than or equal to 2

1 1 | 2

quicksort (0, 3-1) | quicksort (3, 3)

pivot = 1 , l = 1 , r = 2 | r = l = 1

a(1) ⟷ a(2) swap | ∴ return l = 1

a = 1 1 | already sorted.

return l = 1

already sorted

* consider quicksort (4,10) on 4 5 9 3 6 5 3

since a[2] > a[4] pivot = 5 , l=1 , r=7

swap a[1] ↔ a[7] → 3 5 9 3 6 5 4 → a

a[l] < 5 ∴ l → 2
a[r] ≥ 5 ✗ ∴ r = 7

l ≤ r ✓

swap a[2] ↔ a[7] → 3 4 9 3 6 5 5 → a

a[l] < 5 ✓ ∴ l → 3
a[r] ≥ 5 ✓ ∴ r → 4

l < r ✓

swap a[3] ↔ a[4] → 3 4 3 9 6 5 5 → a

a[l] < 5 ✓ ∴ l → 4
a[r] ≥ ✓ ∴ r → 3

l ≤ r ✗

∴ return l = 4 ∴ k = 4

3 4 3 | 9 6 5 5

less than 5 | higher than or equal to 5

quicksort (1, 4-1)

quicksort (4, 7)

a[1] < a[4] ∴ pivot = 4

k = 1 , r = 3

swap a[1] ↔ a[3]

l → 2 , r → 3

a → 3 4 3

swap a[2] ↔ a[3]

a → 3 3 4

l → 3 r → 2

return l = 3

3 3 | 4

quicksort (1,2) → sorted | quicksort (4,4) → sorted

* consider quicksort(4,7) on 9 6 5 5

since a(1) > a(2), pivot = 9. ∴ l = 1, r = 4

swap a(1) ⟷ a(4) → 5 6 5 9

a(l) < 9 ∴ l → 4
a(r) ≥ 9 ∴ r → 3

l ≤ r ✗
∴ return l = 4 ; ∴ k = 4

a → 5 6 5 | 9

quicksort(1,3) on 5 6 5

a(2) > a(1) ∴ pivot = 6, l = 1, r = 3

swap a(1) ⟷ a(3) → 5 6 5

l → 2 r → 3

swap a(2) ⟷ a(3) → 5 5 6
l → 3 r → 2

l ≤ r ✗
∴ return l = 3 . ∴ k = 3

a → 5 5 | 6

quicksort(1,2)

↳ already sorted

↳ return l = 1 ∴ k = 1

a → 5 | 5

quicksort(1,1) quicksort(1,1)

already sorted

quicksort(1,1)

↳ already sorted

quicksort(1,1) → already sorted

∴ by combining all the above divides steps we finally get

1 1 2 3 5 4 5 5 6 9 //

quicksort on → 3 1 4 1 5 9 2 5 5 3

↓

2 1 1 | 4 5 9 3 6 5 3

quicksort on 2 1 1 → 1 1 | 2

quicksort on 4 5 9 3 6 5 3 → 3 4 3 | 9 6 5 5

quicksort on 3 4 3 → 3 3 | 4

quicksort on 9 6 5 5 → 5 6 5 | 9

quicksort on 5 6 5 → 5 5 | 6

by combining we get sorted list.

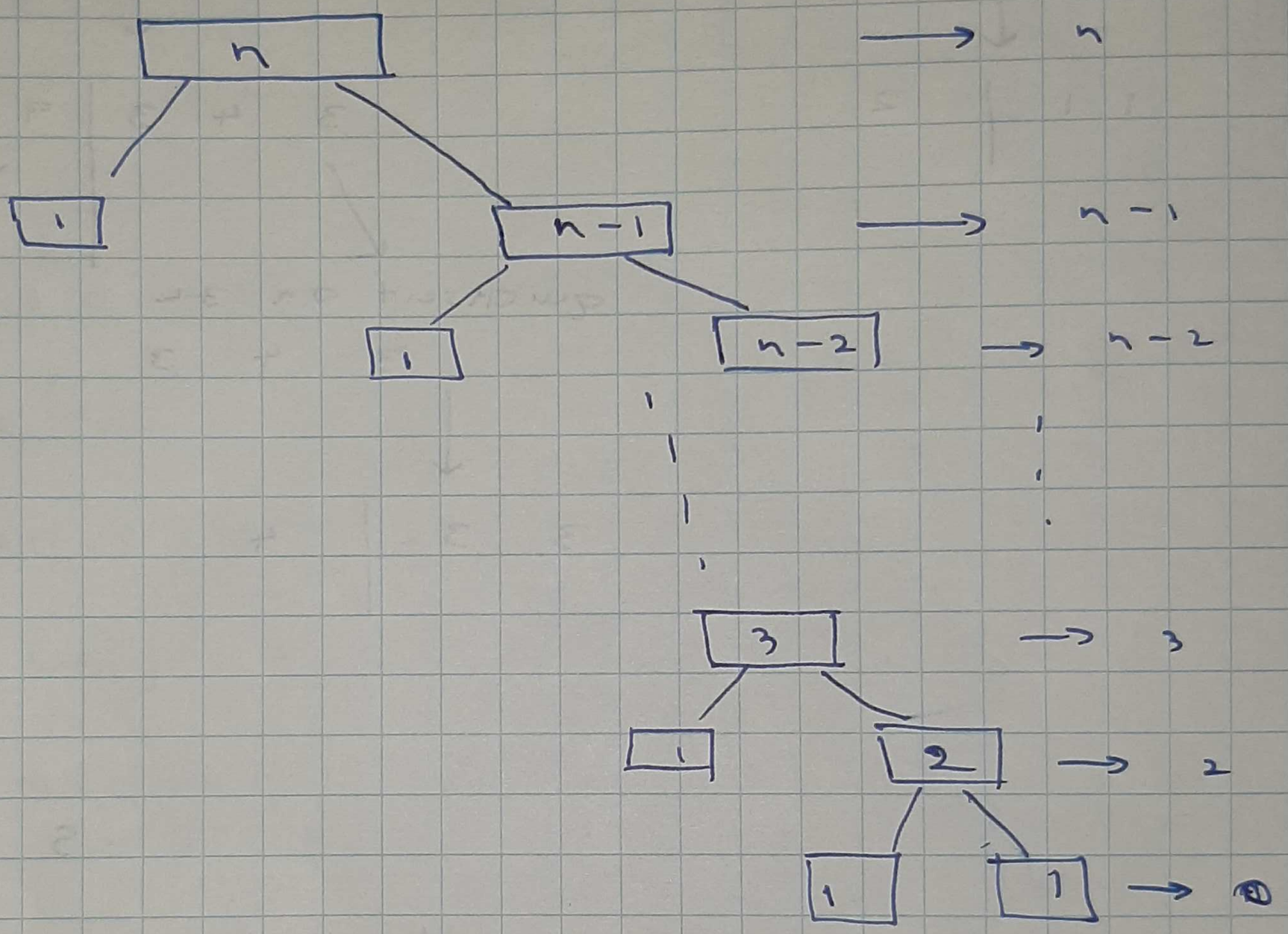
1 1 | 2 3 3 | 4 5 5 | 6 9 //

(C) for the worst case scenario for Quick sort, pivot element must give most unbalance partitions.

In that situation,

$$T(n) = T(n-1) + T(1) + \text{time for partitioning}$$

consrider partioning process,



time complexity of worse case is,

$$T(n) = [n + (n-1) + (n-2) + \cdots 3 + 2] =$$

$$= \left(\frac{n(n+1)}{2} - 1\right)$$

$$= O(n^2)$$

//

∴ worst case complexity for quick sort is in $O(n^2)$ //

(d) Sorting "in place" means sorted items occupy the same space occupied by the original input data set It transform the input into output. A small constant extra space can be used for variables.

(d) Sorting "in place" means sorted items occupy the same space occupied by the original input data set. It transform the input into output. A small constant extra space can be used for variables.
As an example, quick sort algorithm manipulates input into output without using extra space for that manipulation. Therefore quicksort is in-place sort algorithm.

1.6

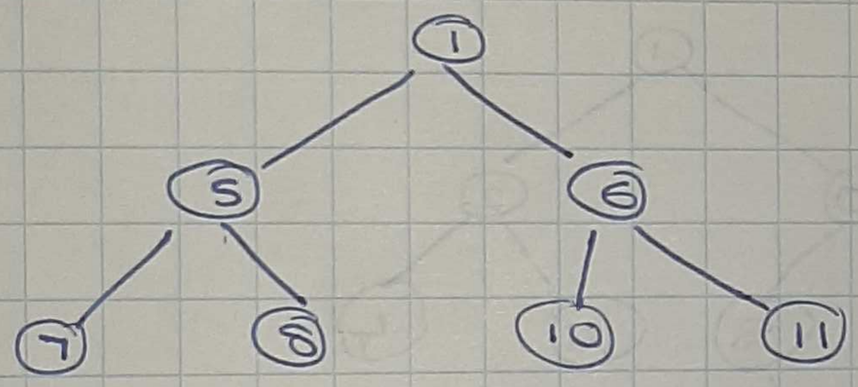
(a) Given heap sort algorithm is an implementation of a priority queue. We use binary heaps to implementation.

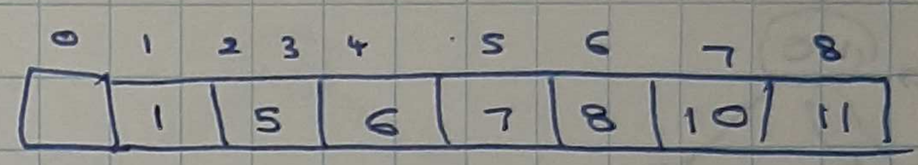
Heap data structure can be implemented using an array. It is implemented as a complete binary tree. for any element i in array, te

left child of binary tree → 2i
right child of binary tree → 2i + 1
parent of binary tree → $\lfloor i/2 \rfloor$

for the heap creation, maximum number of elements are needed to know beforehand.

ex:- heap in array form



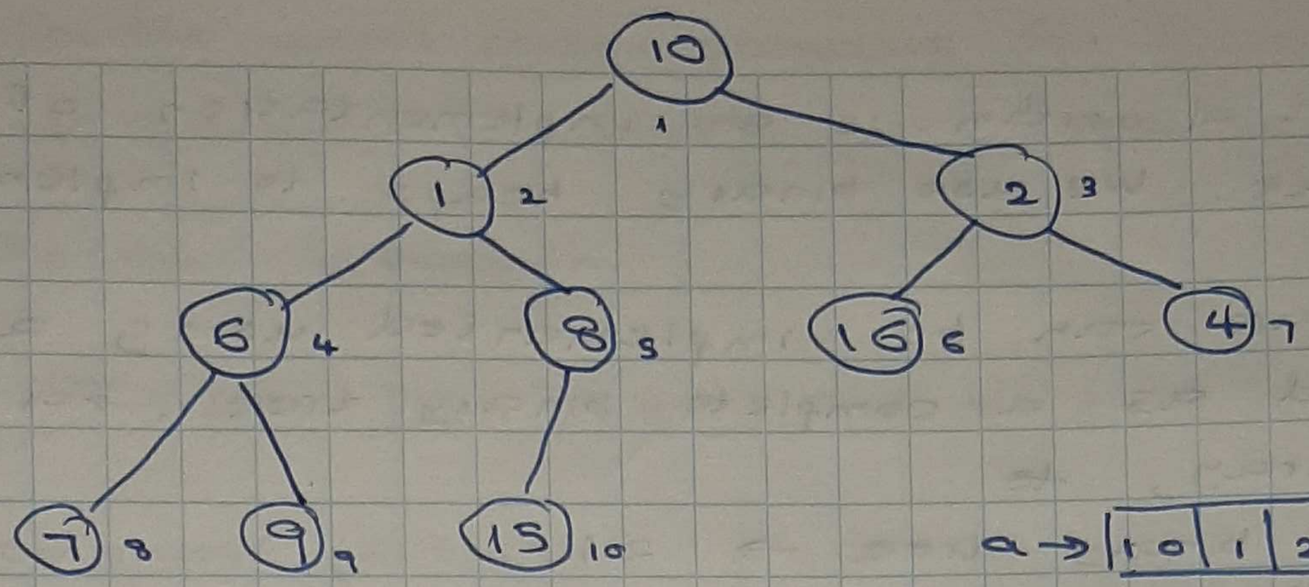


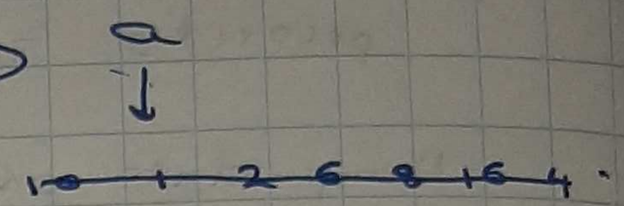
0th position of array is reserved for special purposes.

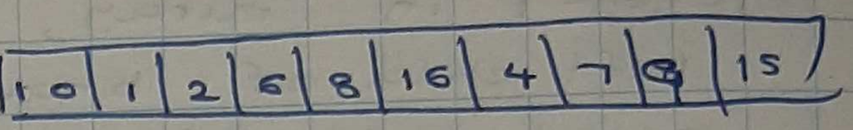
In heaps, heap order property is prescived at all times, which is for every node x, key of parent node is small or equal to node x. exception of the root.

given sorting algorithm utilizes a min heap. During sorting, ist heap is constructed and then minimum value in heap is taken each time while preserving the heap order property.

(b)

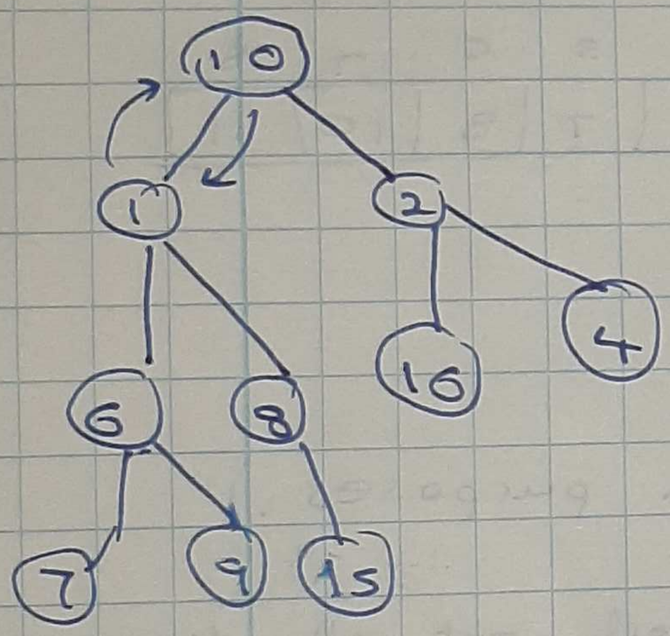


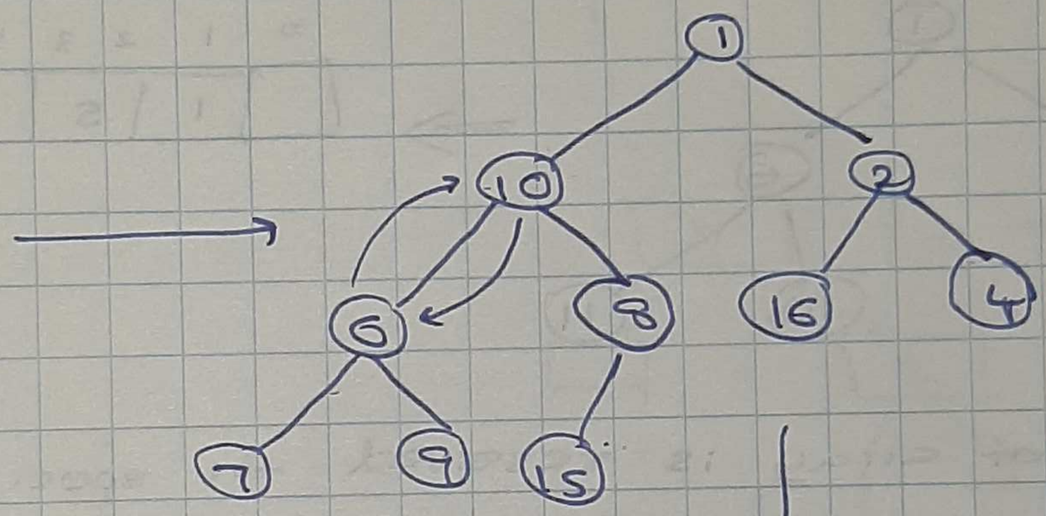


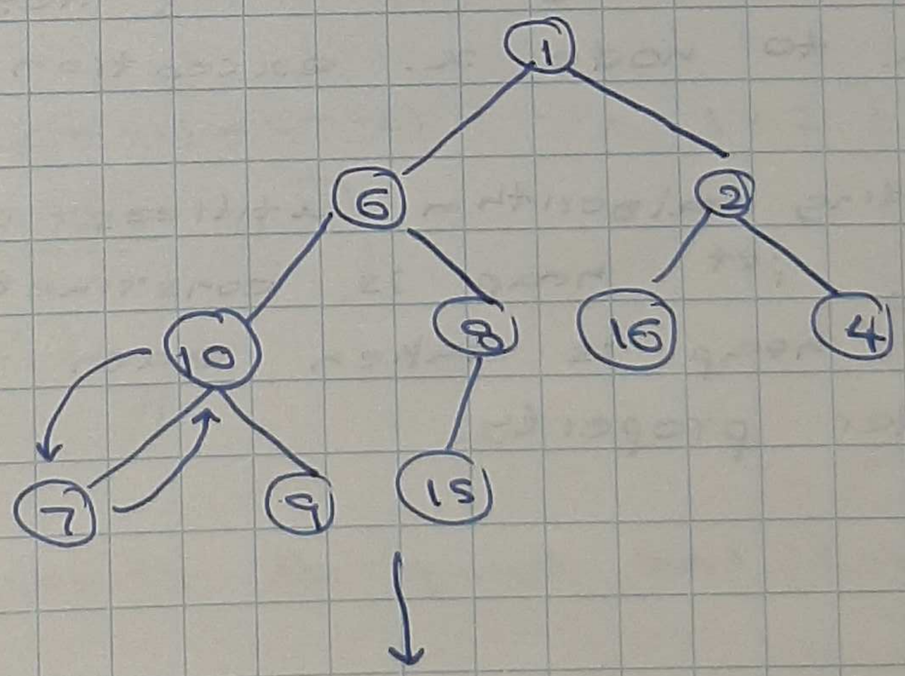


- This data structure does not follow heap order property. a[2] and a[3]'s ~~child~~ parent node has a laiger value.

- we need to call "pushdown" function to ensure heap order property.



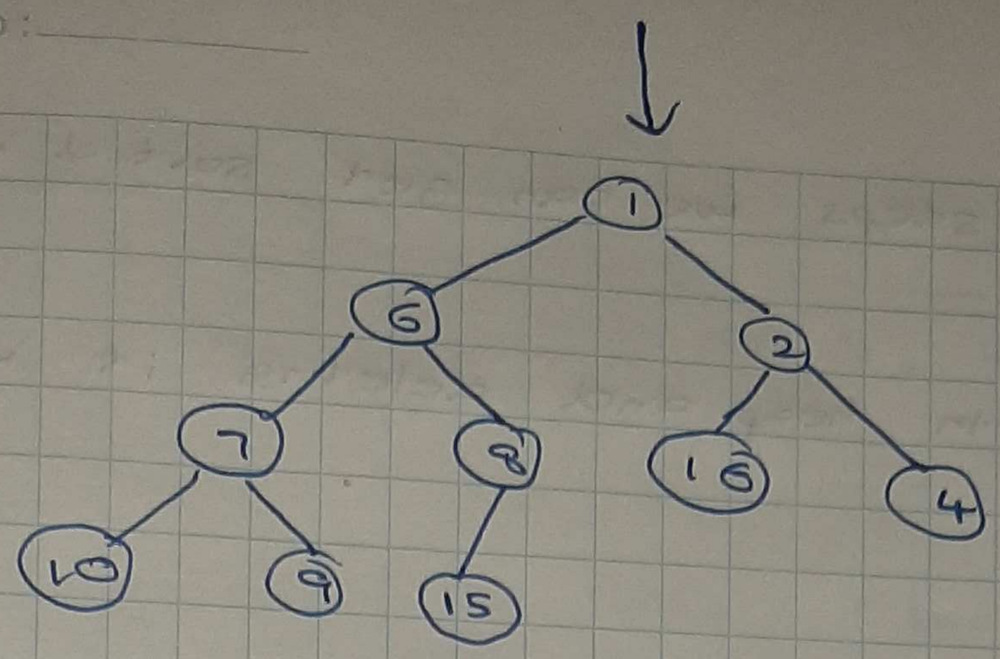




(15)

No :

Date: __/__/____

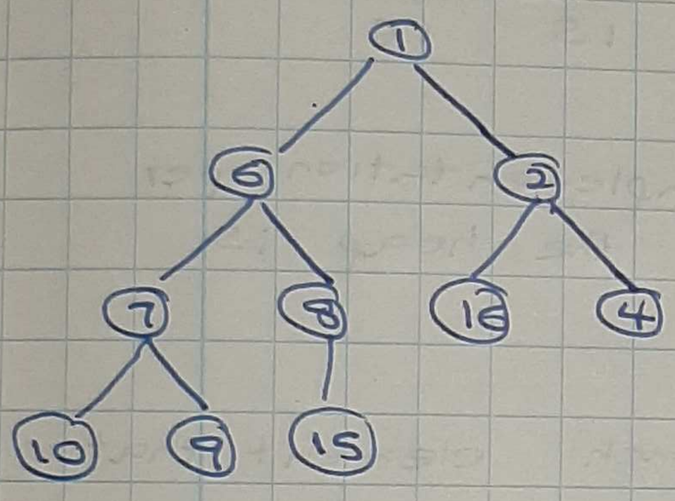


In matrix form → a = ← 0th position reserved

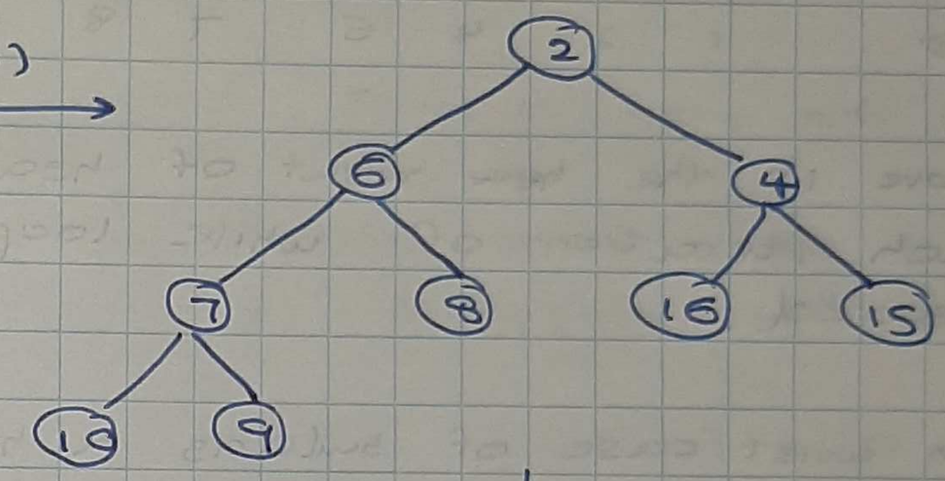
0	1	2	3	4	5	6	7	8	9	10	11
0	1	6	2	7	8	16	4	10	9	15	

Now this heap structure follows heap order property.

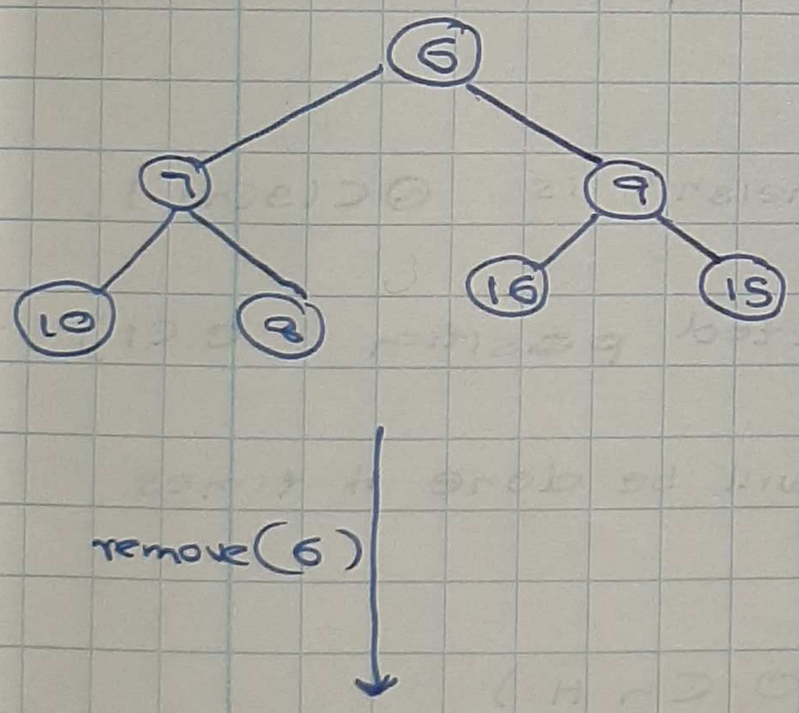
Now we can use heapsort algorithm and sort this.



remove(1)

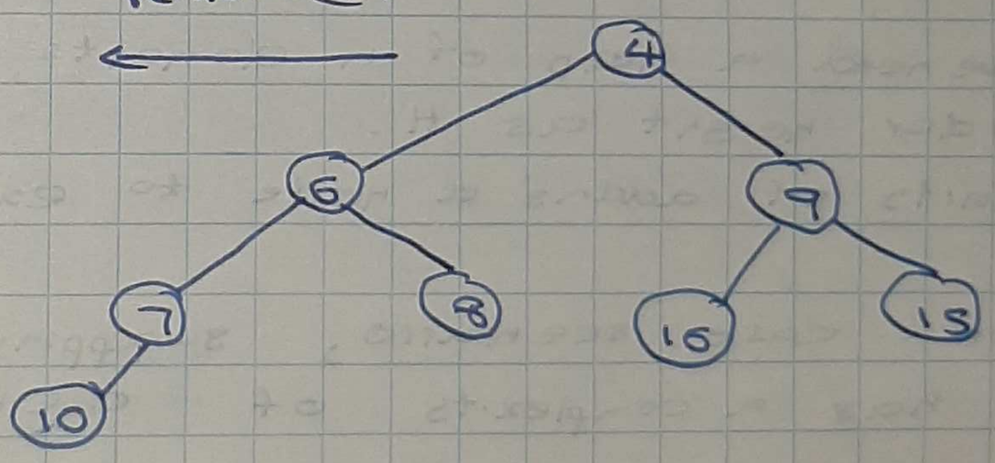


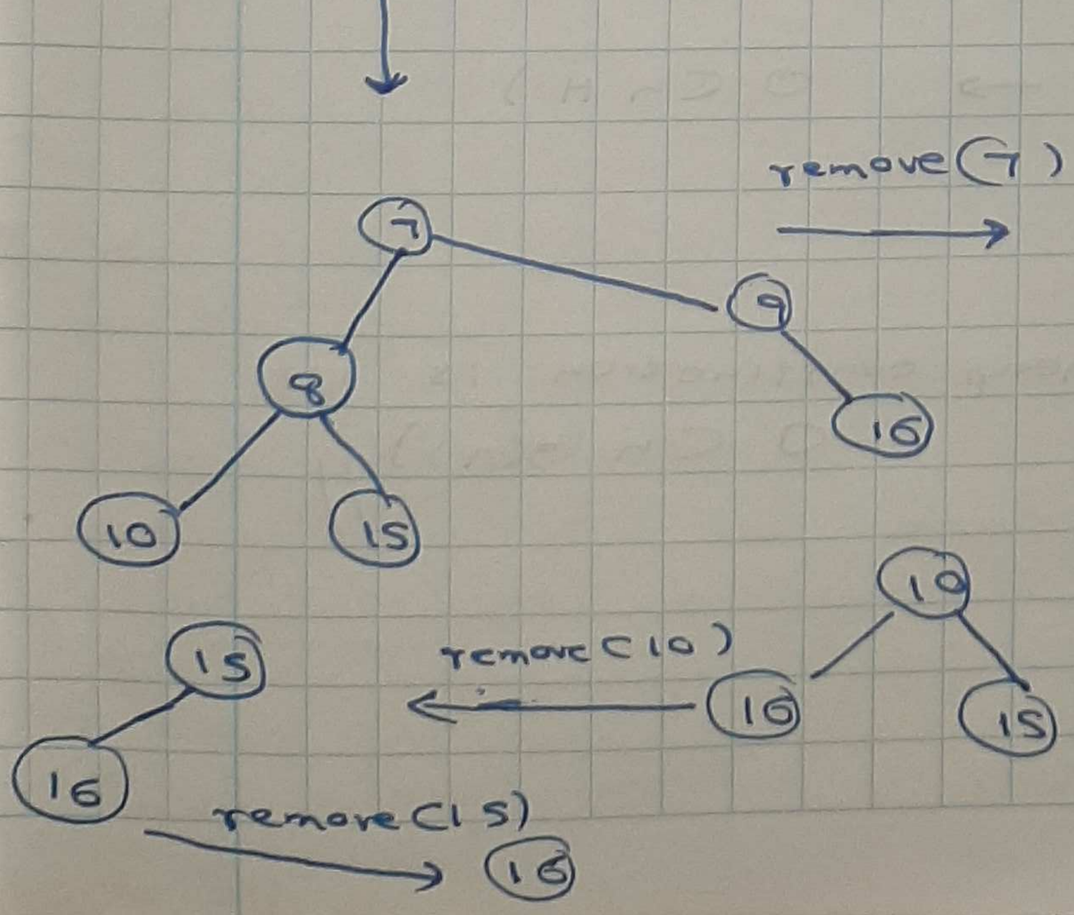
remove (2)

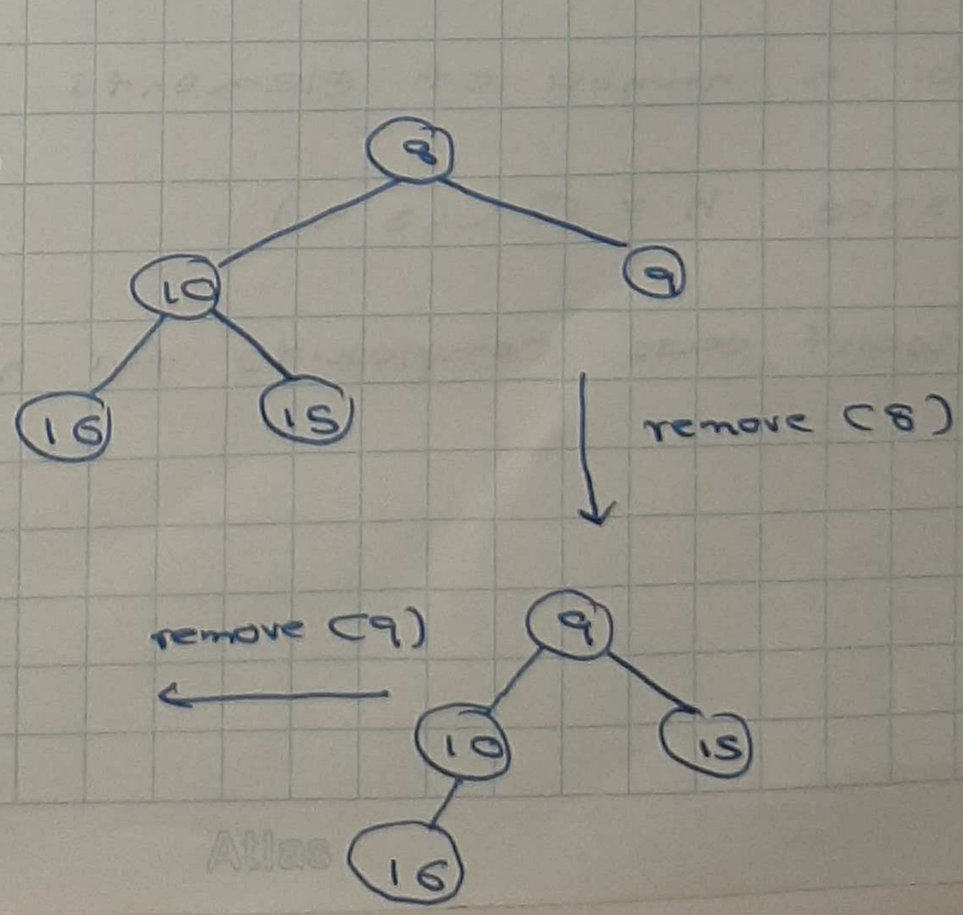


remove(6)

remove(4)







by following above steps we can get sorted numbers.
by executing,

getting minimum in heap and deleting it until the heap is empty.

1 → 1
2 → 1 2
3 → 1 2 4
4 → 1 2 4 6
5 → 1 2 4 6 7
6 → 1 2 4 6 7 8
7 → 1 2 4 6 7 8 9
8 → 1 2 4 6 7 8 9 10
9 → 1 2 4 6 7 8 9 10 15
10 → 1 2 4 6 7 8 9 10 15 16

above is the ~~hea~~ result of heap sort implementation per each iteration of while loop until the heap is cleared.

(c) for worst case of building a heap, each element that inserted needs to be the new minimum of the heap. therefore, each time it needed to be moved to root by swaping with other nodes.

- If we need a heap of n elements, height is $\Theta(\lg(n))$. consider height as H.

complexity of adding a node to expected position : $O(1)$

for worst case scenario, swapping will be done H times.
this has a complexity of $O(H)$

for n number of elements → $O(nH)$

since $H = \Theta(\lg n)$

worst case complexity for heap construction is in $O(n \lg(n))$ //

(d) while (S is not empty)
{
y = min (S); — c_1
print the value of y; — c_2
delete y from S; — c_3
}

for n elements in heap, loop will iterate n times, for each iteration, 3 statements inside the loop will be executed once.

- Since heap order poperty is preserved, extracting minimum value y is straight forward.
- But delete operation requires comparisons between child nodes and selecting the smaller child and moving it to the root and also doing other operations to preserve heap order property.

If a node is to be deleted from a heap of height H,

- complexity of swaping parent and leaf node : O(1) but for worst case scenario, swapping will be done H times.

∴ total complexity is $O(1) + O(H) = O(H)$

but $H = O(\lg n)$

complexity per single operation is $O(\lg n)$

for n operations,

complexity is $O(n.\lg(n))$ //