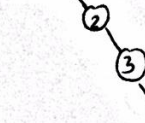


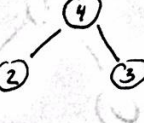
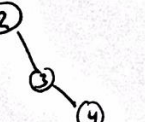


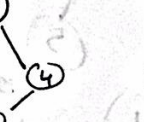
1.  1234

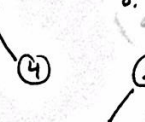
2.  1243

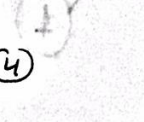
3.  1234


4.  1243

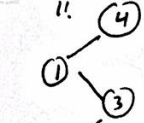
5.  1234


6.  1243


7.  1234


8.  1243

9.  1234

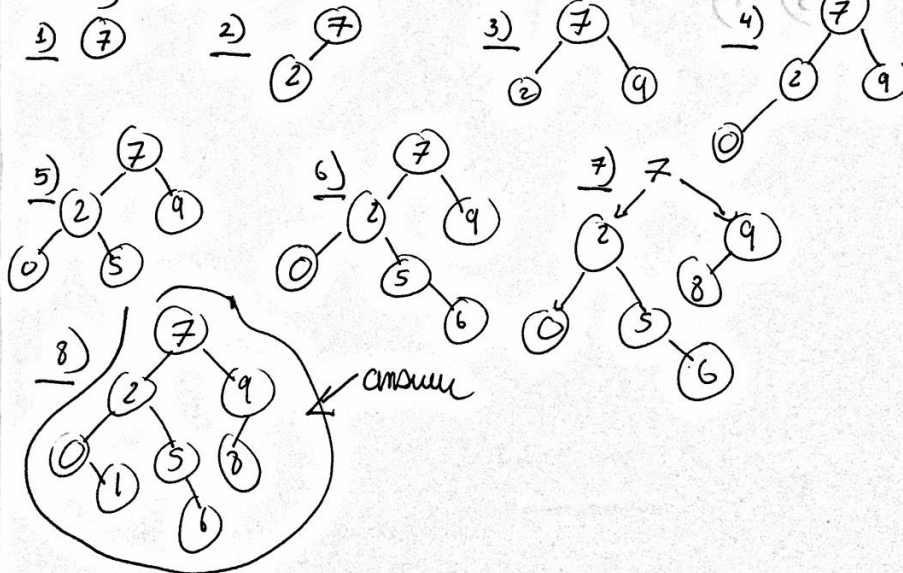
10.  1234

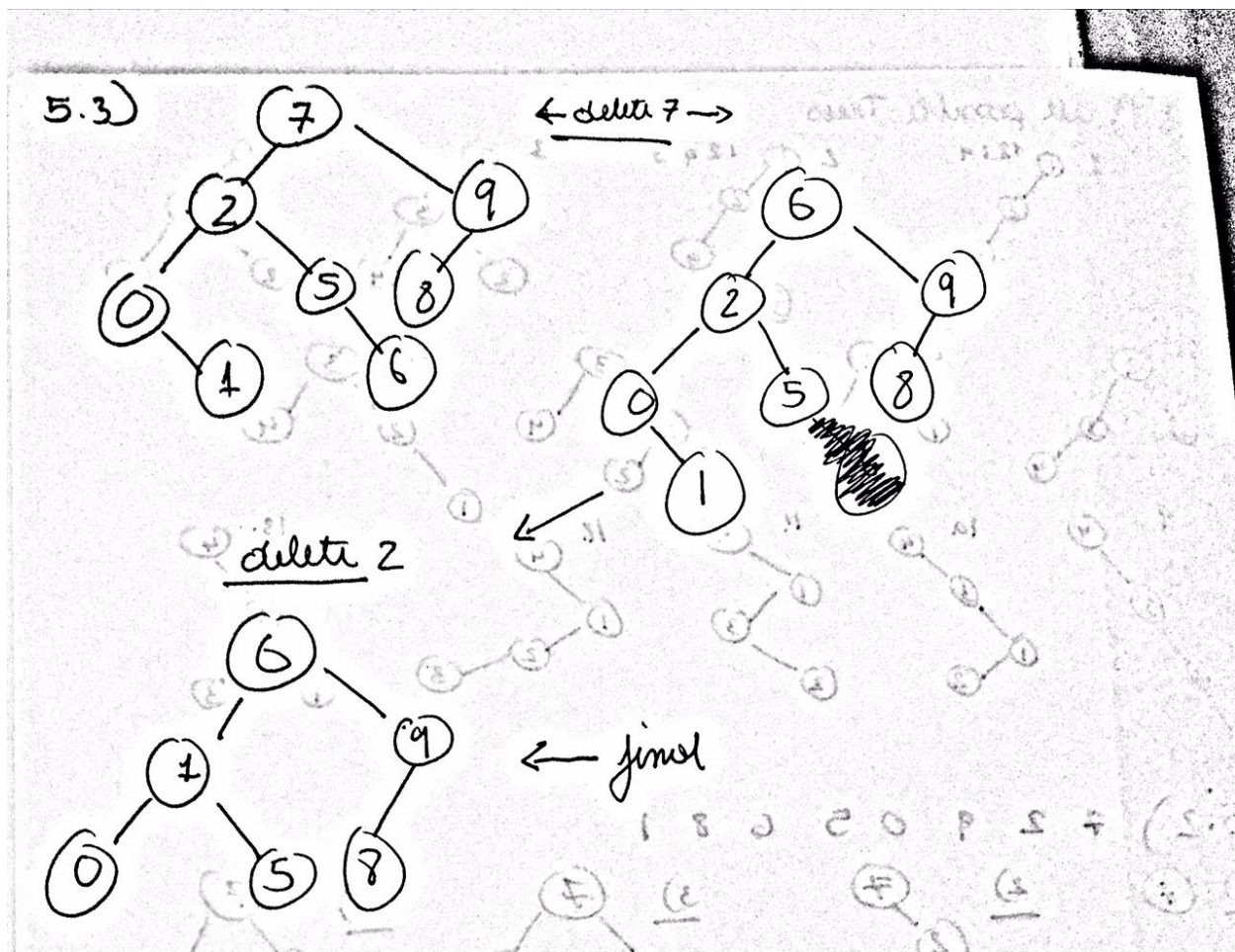
11.  1243

12.  1243

13.  1243

5.2) 7 2 9 0 5 6 8 1





5.4

By use of the deletion algorithm in figure 5.5, the order of deletion would have no effect on the outcome of the tree because the `deletemin()` method is called on the right most subtrees/descendant nodes of any node with 2 children. Thus irrespective of the order in which we delete the minimum will always be returned in both cases thus having no true effect on the configuration of our tree.

5.5

Size of Array	Run Time T(n)	Run Time T(n^2)	Run Time T(n)/n	Run Time T(n^3)
10	0.13642882500425912	0.018612824292042758	0.013642882500425913	0.0025393257481741245
25	0.0731366449908819	0.0053489688405222915	0.0029254657996352763	0.00039120563515656807
50	0.0738092900137417	0.005447811292332631	0.0014761858002748341	0.0004020990836159161
75	0.06942811899352819	0.00482026370697951	0.0009257082532470425	0.00033466184222835866
100	0.07210543399560265	0.00519919361169421	0.0007210543399560265	0.00037489011179837585
500	0.07524277002085	0.005661474440410524	0.0001504855400417	0.0004259850192987295
300	0.07641168200643733	0.005838745147052899	0.0002547056066881244	0.0004461483374932353
800	0.0723230819858145	0.005230628187926845	9.040385248226812e-05	0.00037829515127274553
1000	0.07253330701496452	0.0052610806265271015	7.253330701496452e-05	0.00038160357631437215
3000	0.08027870798832737	0.006444670956275137	2.6759569329442456e-05	0.0005173698577796663
5000	0.07643776398617774	0.005842731763206611	1.528755279723555e-05	0.0004466053515505311
8000	0.0725587579945568	0.00526477336171266	9.0698447493196e-06	0.00038200541624869815
10000	0.0722106659959536	0.00521438028357917	7.221066599595361e-06	0.0003765338730334213
20000	0.07202973199309781	0.0051882822909974985	3.6014865996548907e-06	0.00037371058292508534
30000	0.07224866599426605	0.005219869737951016	2.408288866475535e-06	0.0003771286252308
40000	0.07669677201192826	0.005882394837049702	1.9174193002982066e-06	0.0004511606957013449
50000	0.07536778500070795	0.005680303015912939	1.507355700014159e-06	0.00042811185644219934

Make heap has a linear strict upper bound or $O(n)$

5.6

A heap is represented by a binary tree

To build a heap, a binary tree structure must be employed thus, given n values to build a heap, we must first describe deduce the path lengths of the root's rightmost and leftmost nodes.

Path lengths can be calculated by averaging for all i between 0 and $n - 1$ the sum $\frac{i}{n}(p(i) + 1) + \frac{n-i-1}{n}(p(n-i-1) + 1) + \frac{1}{n}$ consisting of the average path length of the left subtree by its size, the average length of the right subtree weighted by its size and the root's contributions

$$\begin{aligned}
 & \frac{1}{n} \sum_{i=0}^{n-1} i(p(i) + 1) + \frac{1}{n} \sum_{i=0}^{n-1} (n-i-1)(p(n-i-1) + 1) + \frac{1}{n} \rightarrow p(n) \\
 & = 1 + \frac{1}{n^2} \sum_{i=0}^{n-1} i(p(i) + (n-i-1)p(n-i-1)) \rightarrow p(n) \\
 & \leq 1 + \frac{2}{n^2} \sum_{i=0}^{n-1} 4i \log(i) + i \leq 1 \\
 & + \frac{2}{n^2} \sum_{i=1}^{n-1} 4i \log(i) + \frac{2}{n^2} \sum_{i=1}^{n-1} i \leq 2 + \frac{8}{n^2} \sum_{i=1}^8 i \log(i) \rightarrow p(n) \\
 & \leq 2 + \frac{8}{n^2} \left(\frac{n^2}{2} \log(n) - \frac{n^2}{8} \right) \leq 1 + \log(n) = O(\log n)
 \end{aligned}$$

height of tree is $\log n$ and a heap size of n has $\frac{n}{2^{h+1}}$ nodes meaning

the run time depends on the height of the tree. Tree height can be expressed as $\log(n)$ where n is the number of nodes. We must sum from node 0 to the maximal height of the tree of which is $\log(n)$

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log(n)} \left(\frac{n}{2^{h+1}} \right) * O(h) = O \left(n * \sum_{i=0}^{\log(n)} \left(\frac{h}{2^h} \right) \right) = O \left(\left(n * \sum_{i=0}^{\log(n)} \left(\frac{h}{2^h} \right) \right) \right) \\
&\rightarrow \left(n * \sum_{i=0}^{\log(n)} h * \left(\frac{1}{2^h} \right)^h \right) \\
&\rightarrow \sum_{n=0}^{\inf} \frac{1}{1-x} \rightarrow \sum_{n=0}^{\inf} nx^n = \frac{x}{(1-x)^2} \rightarrow O \left(n * \frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2} \right) = O(n * 2) = O(n)
\end{aligned}$$

Time to create a tree = $O(n)$ and time to extract from a tree is $n * \log(n)$ thus the total time is $O(n) + O(n * \log(n)) = \max(O(n), O(n * \log(n))) = O(n * \log(n))$

Time complexity is $O(n * \log n) \rightarrow$ space complexity
= size of the elements to sort and the size of the heap = $2 * n = O(2n) = O(n)$

5.7

```

Ptrs = my_data_object();
Value = heap()
Head = first()
I = 0
While head != null
    Ptrs[i] = head
    Value.push(head.height())
    Head = next(head)
    I++
End
Temp = impressive(value)
For I=0, I < length(temp), i++
    Ptrs[i].computedValue = Temp[i]
End
➔ Run time is  $O(3n) \rightarrow O(n)$ 

```

5.8

To push a new element = $1 * \text{push}()$

To push n elements to stack = $n * \text{push}()$

To push stack 1 to stack 2 = $n * \text{pop}() + n * \text{push}()$

To pop from stack 2 = $n * \text{pop}()$

To insert

1 item = $O(1)$

N item = $O(n)$

To remove

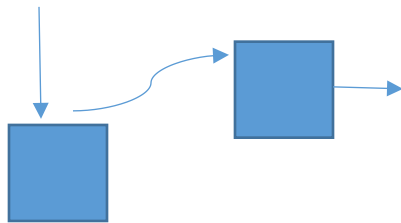
1 item = $O(1)$

N item = $O(n)$

Space complexity of queue

→ Push n items into stack 1 + Pop n items from stack 1 + Push n items into stack 2 + Pop n items from stack 2

→ $4 * O(n) \rightarrow O(n)$



5.9

```
V = heap()
```

```
For item in server_checksum
```

```
    For value in client_checksum
```

```
        If value is not equal to item
```

```
            [bmin, bmax] = getBlockFromChecksum(item)
```

```
            V.push([bmin, bmax])
```

```
        end
```

```
    end
```

```
end
```

```
while empty(V) is false
```

```
    print("blocks are %i" ,V.pop(i));
```

```
end
```

```
Time complexity  $\rightarrow O(\text{item} * \text{value})$ 
```