Marwan Jabbour

40151859

Comp 352

Assignment 3

Friday, June 12 2020

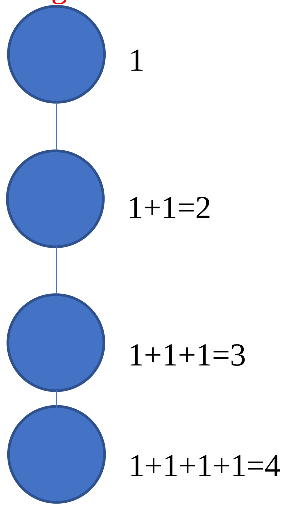
Question 1

a)

```
Algorithm depth (position)
Input: Position position in a tree T
Output: Depth of position
    If isRoot(position)    then
        return 0
    else
        return 1+depth(parent(position))
```

```
Algorithm ComputeDepths(T)
Input: Tree T with n nodes
Output: Depths of all nodes of a tree T
    temp_iterator←new iterator of all positions() in tree T
    for each Position position in tree T do
        Print the element stored in the node (this step is unnecessary, but clearer to user)
        Call method depth on the node
```

Time Complexity: Let n be the number of nodes of T. The time complexity of depth algorithm depends on the depth of the position. It runs in O(d+1) time where d is the depth of pos in T. It is a recursive algorithm. The worst-case scenario for depth would be if we tried to calculate the depth of the most external node with depth n-1. This would result in depth running in O(n) time. This worst-case scenario would occur if the nodes were organized in one single long branch.
In that case, the method ComputeDepths would run 1+2+3+n times, giving a time complexity of O(n^2). To understand this better, refer to the figure below where the number of operations are listed for each node in this hypothetical tree.
Space Complexity: (*I am not sure if this part is needed, so if not kindly skip it*). The space complexity of my algorithm is also O(n) because we need to allocate space for every node when using the iterable collection positions().



b)

Algorithm Count-Full-Nodes(t)

Input: Binary search tree rooted at t

Output: Number of full nodes in the tree

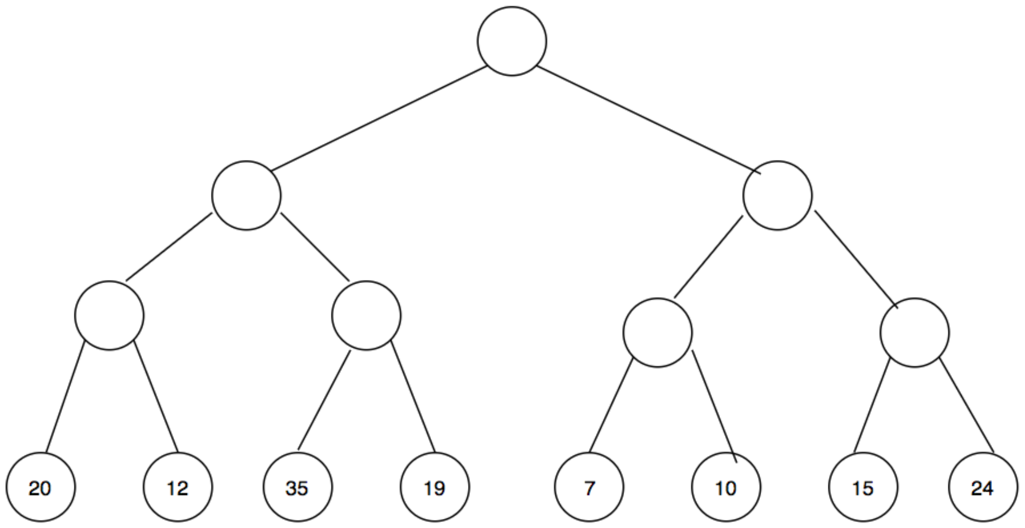
```
temp_queue ← new queue of type Node
if  t ≠ null then
    add t to temp_queue
count_full ← 0
while ! temp_queue.isEmpty() do
    temp_node ← temp_queue.dequeue()
    if  temp_node.right ≠ null ∧ temp_node.left ≠ null  then
        increment count
    if temp_node.left ≠ null  then
        add temp_node.left to temp_queue
    if temp_node.right ≠ null  then
        add temp_node.right to temp_queue
return count_full
```

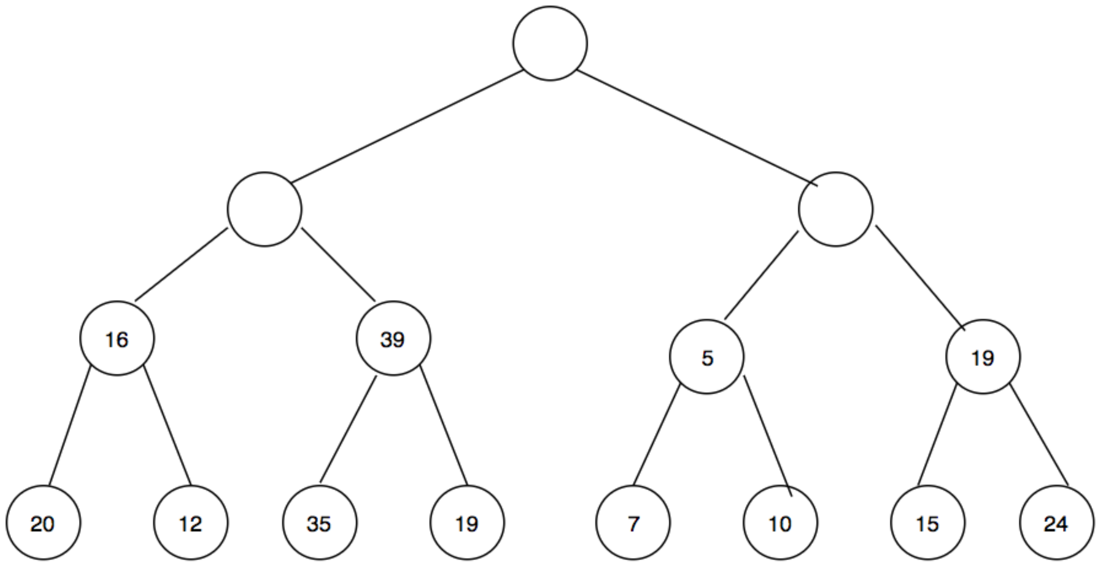
Time Complexity: Let n be the number of nodes in the binary search tree. In the while loop, we will have to check every single node in the tree. The operations inside the while loop all run in O(1) time, and since we have n nodes in the tree, the algorithm runs in O(n) time.

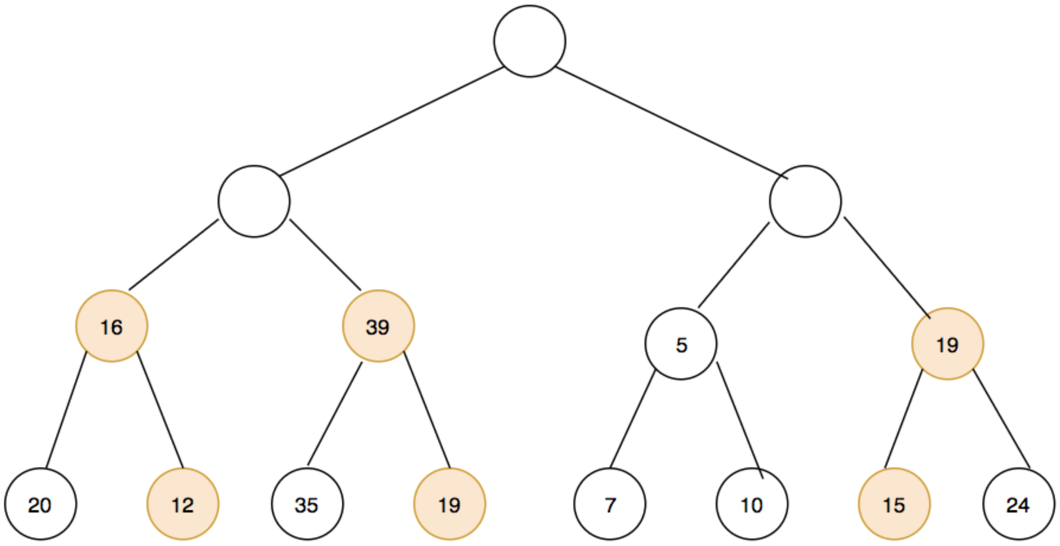
Space Complexity: (*I am not sure if this part is needed, so if not kindly skip it*). The space complexity of my algorithm is also O(n) because we have to allocate n spaces in the temp_queue for every node in the tree.

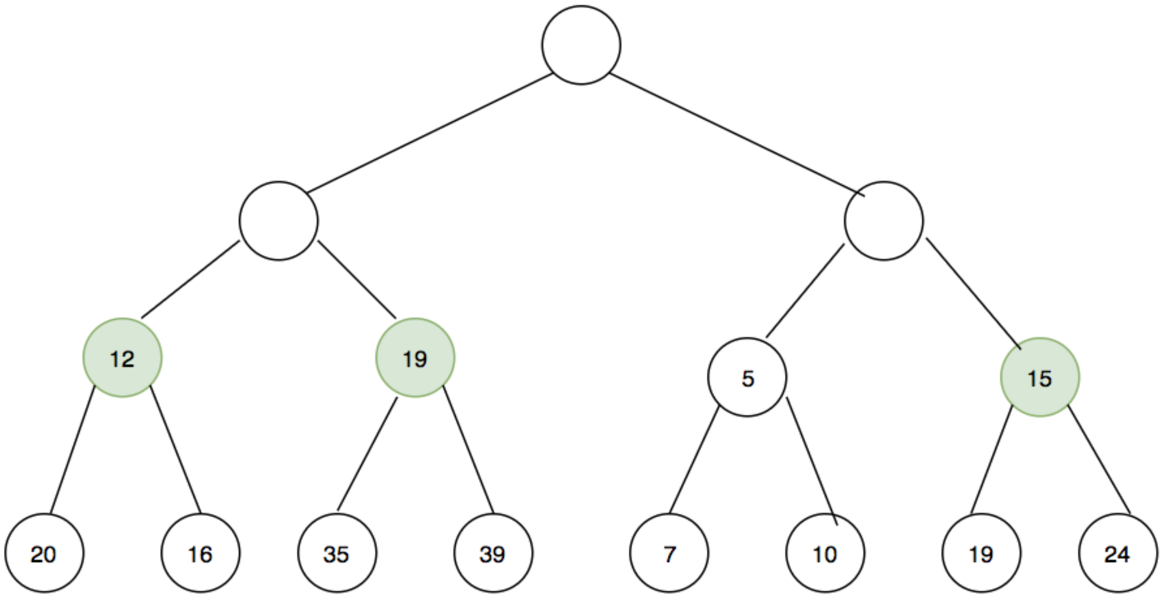
Question 2

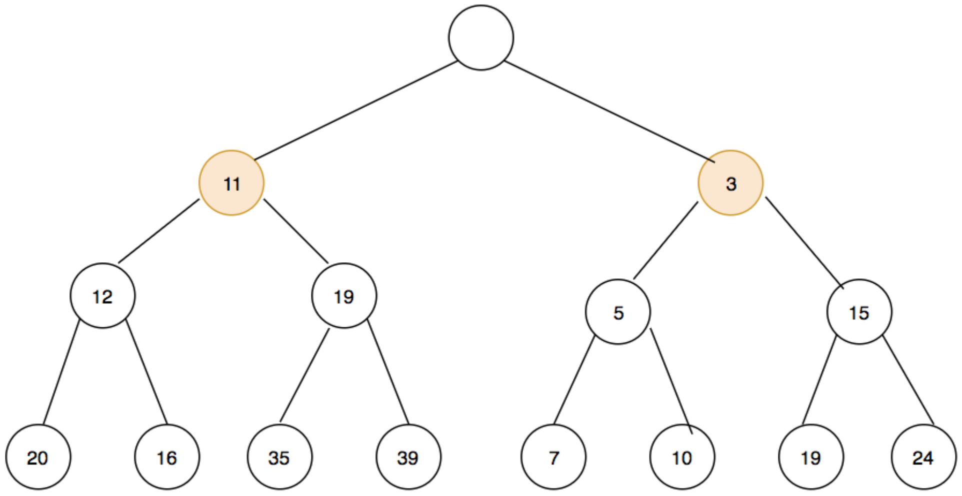
a)

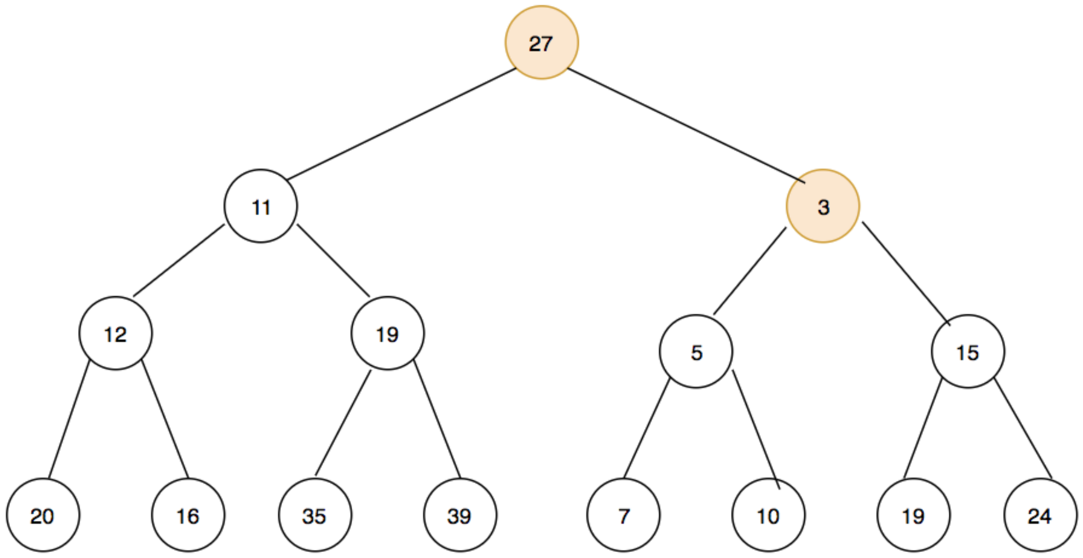


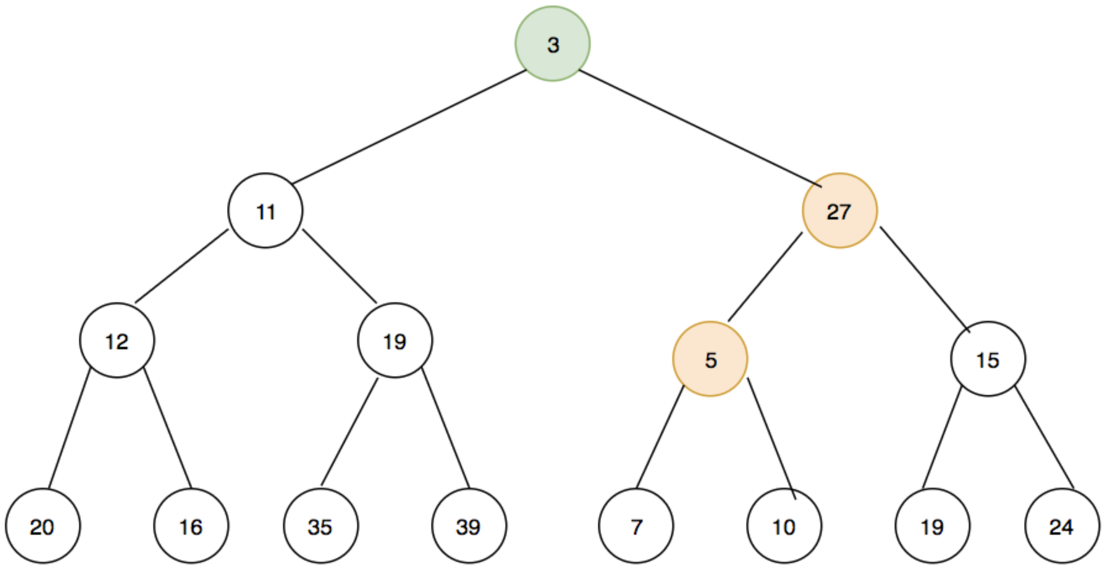
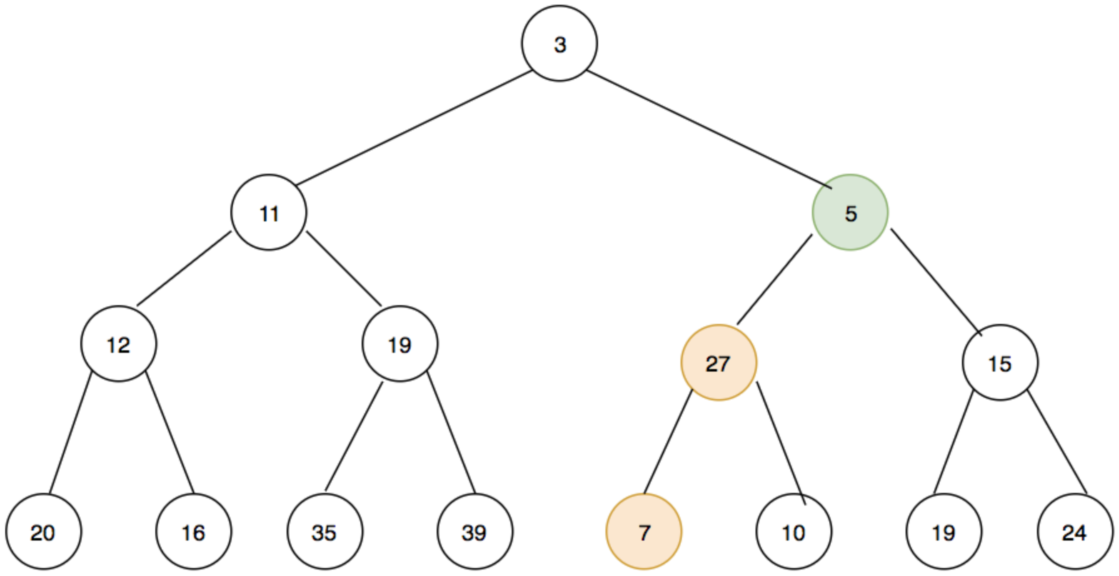


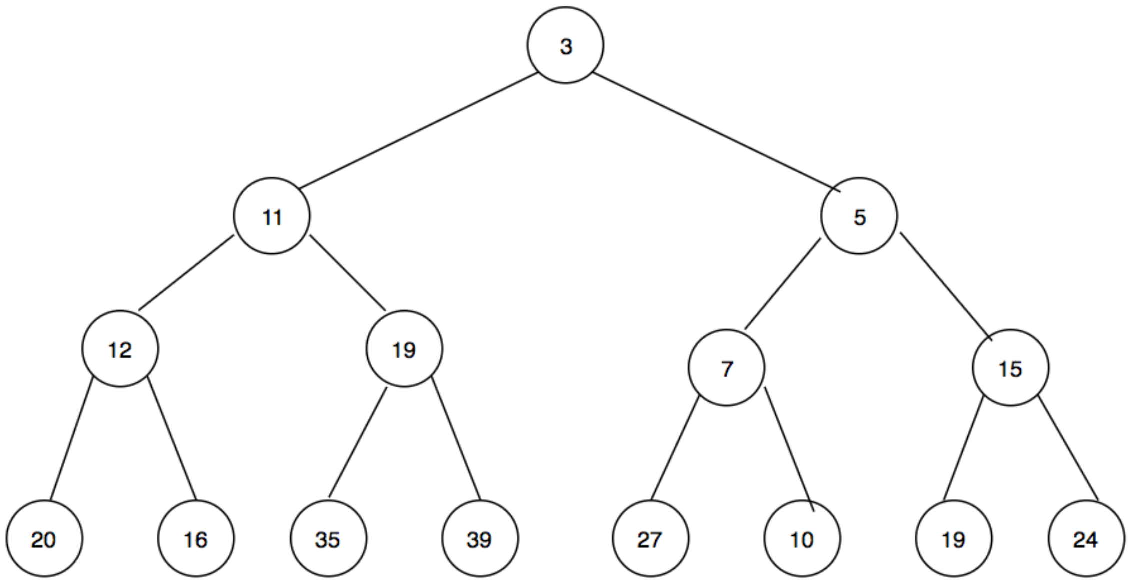

16
39
5
19
20
12
35
19
7
10
15
24


12
19
5
15
20
16
35
39
7
10
19
24

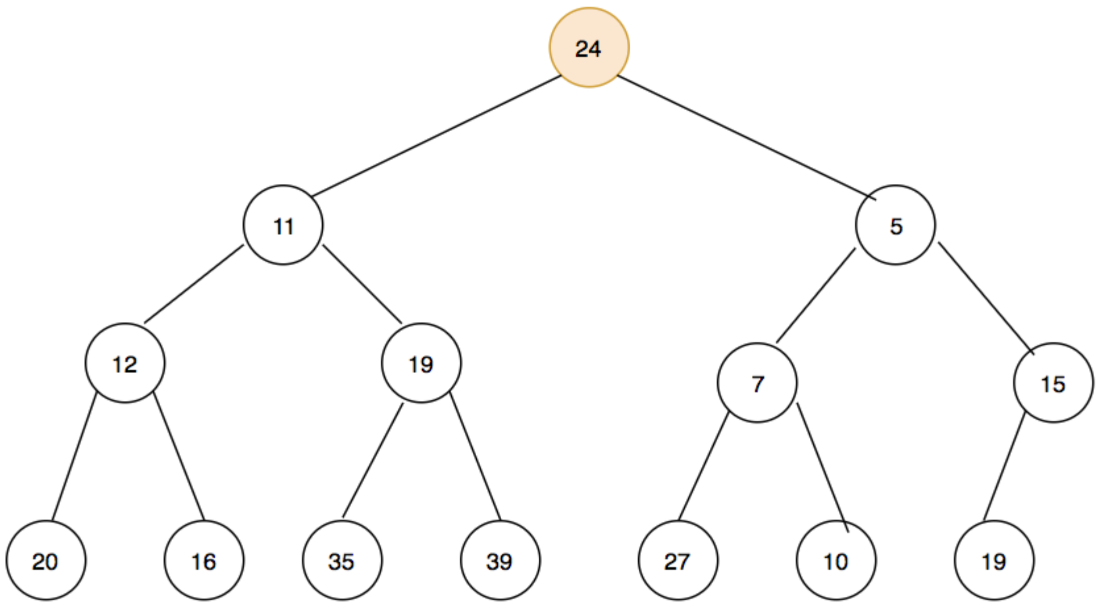

11
3
12
19
5
15
20
16
35
39
7
10
19
24

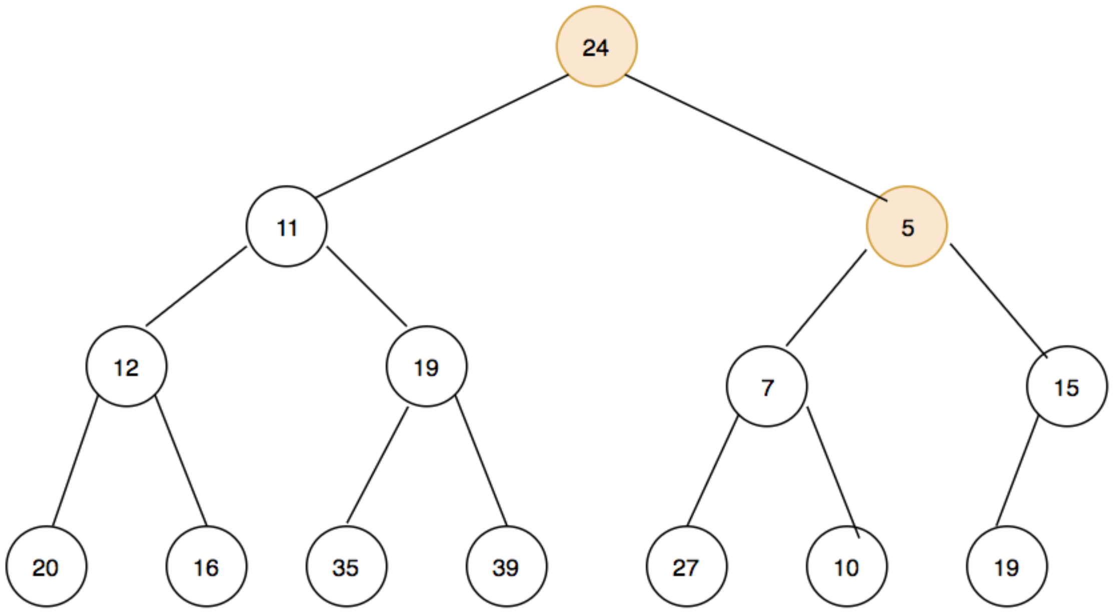

27
11
3
12
19
5
15
20
16
35
39
7
10
19
24

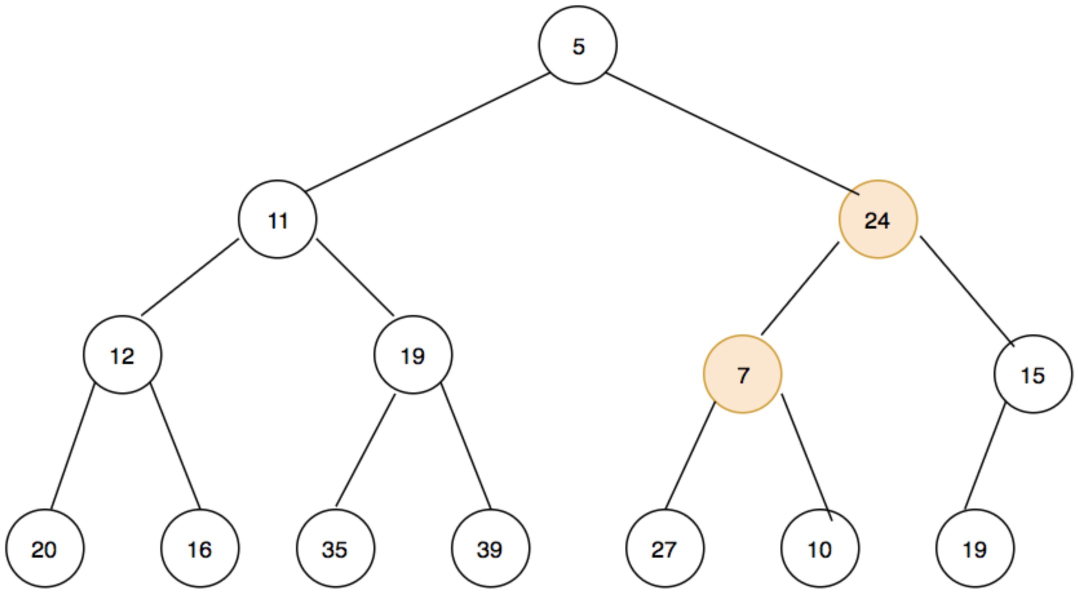

3
11
27
12
19
5
15
20
16
35
39
7
10
19
24

3
11
5
12
19
27
15
20
16
35
39
7
10
19
24

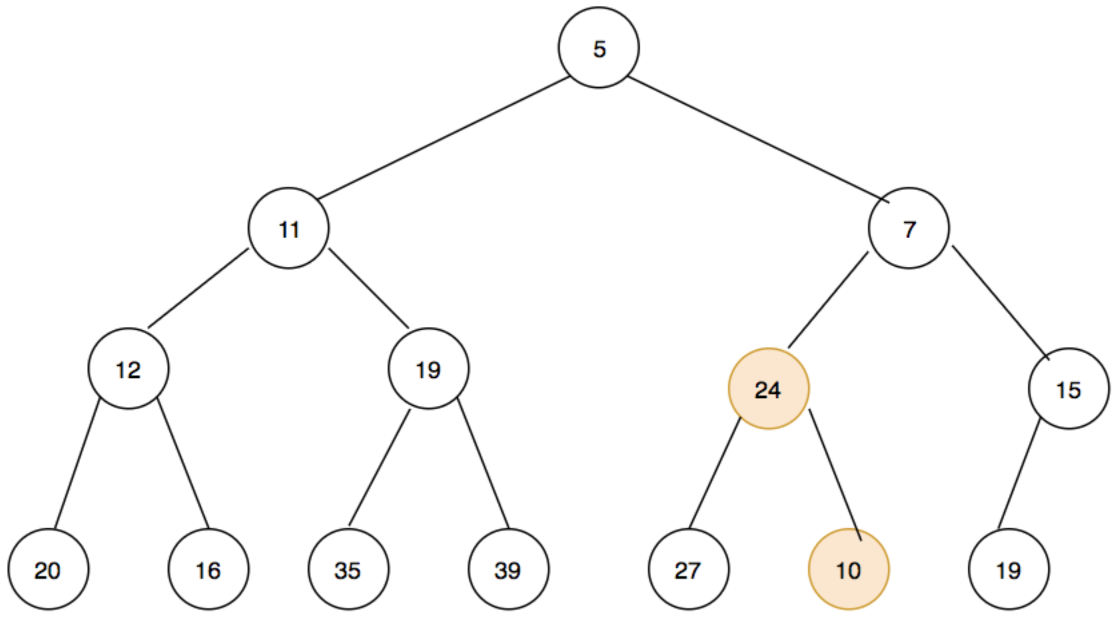

3
11
5
12
19
7
15
20
16
35
39
27
10
19
24

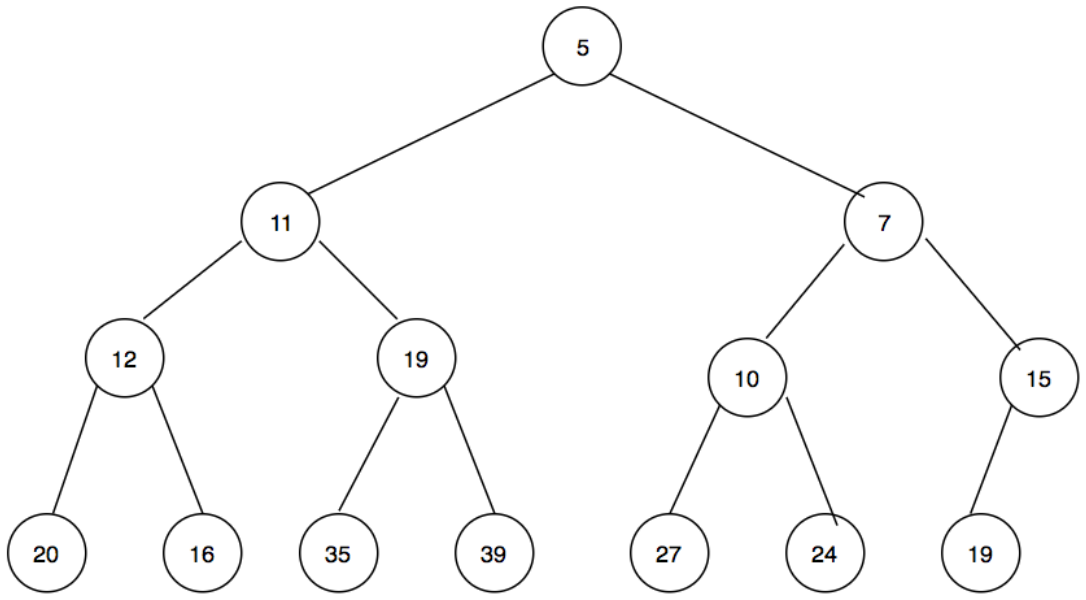
removeMin: part one



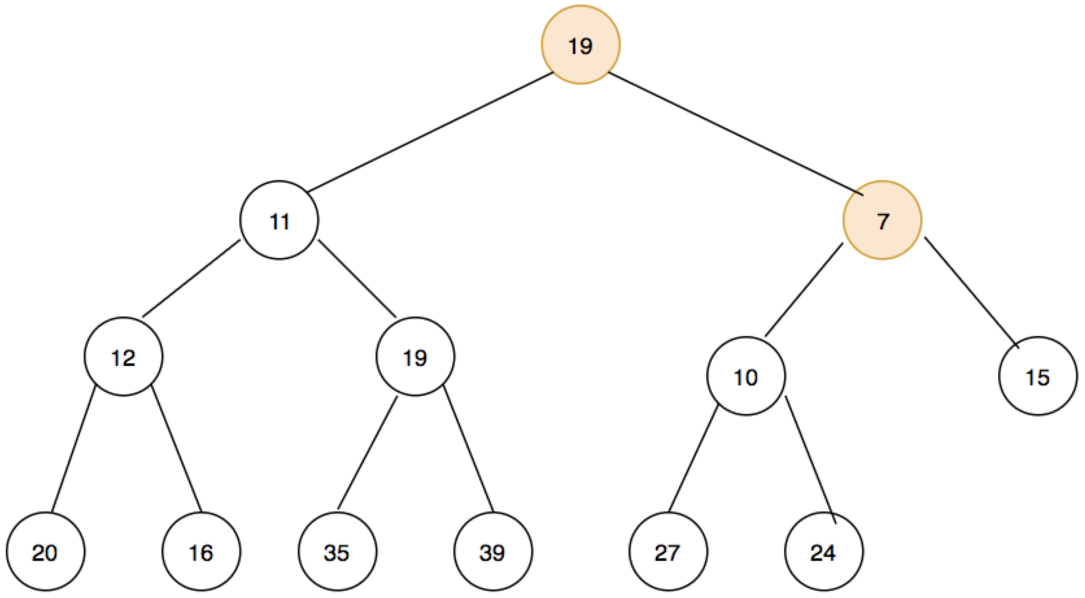

24
11
5
12
19
7
15
20
16
35
39
27
10
19

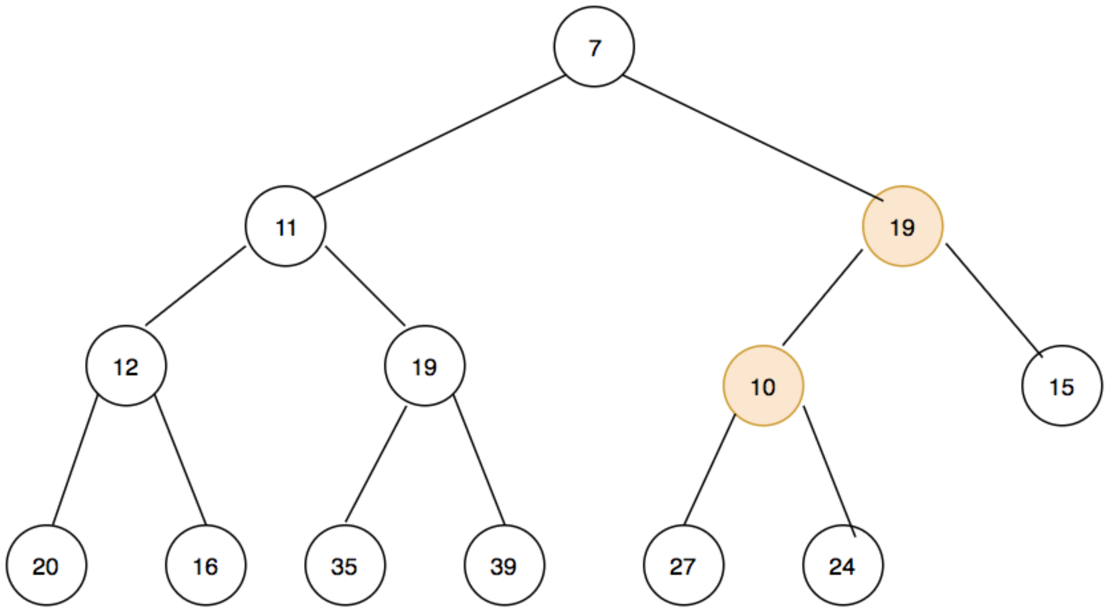

5
11
24
12
19
7
15
20
16
35
39
27
10
19

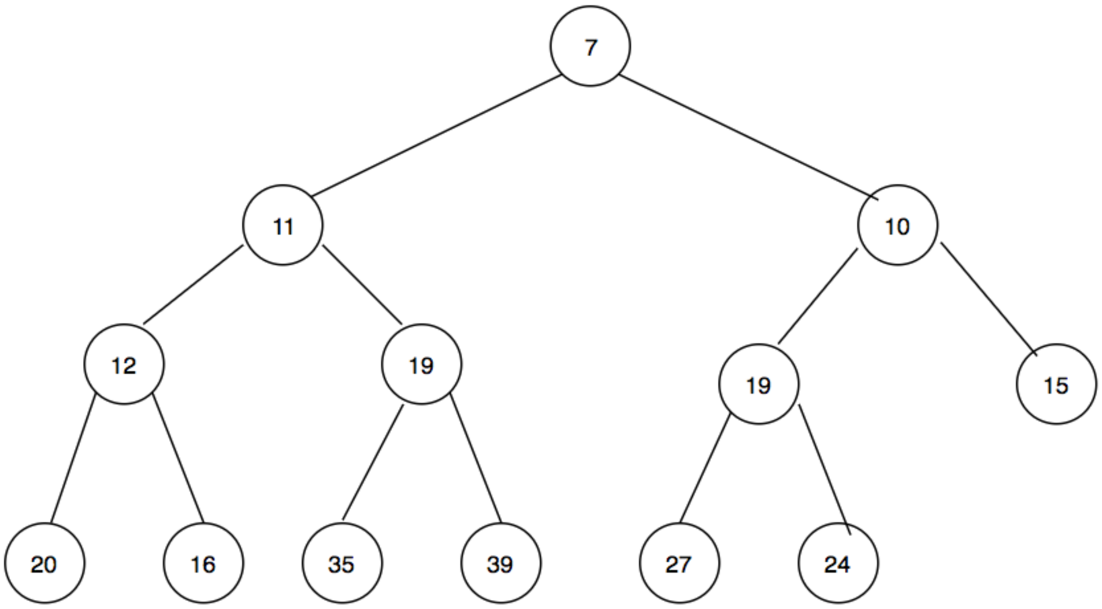

5
11
7
12
19
24
15
20
16
35
39
27
10
19


5
11
7
12
19
10
15
20
16
35
39
27
24
19

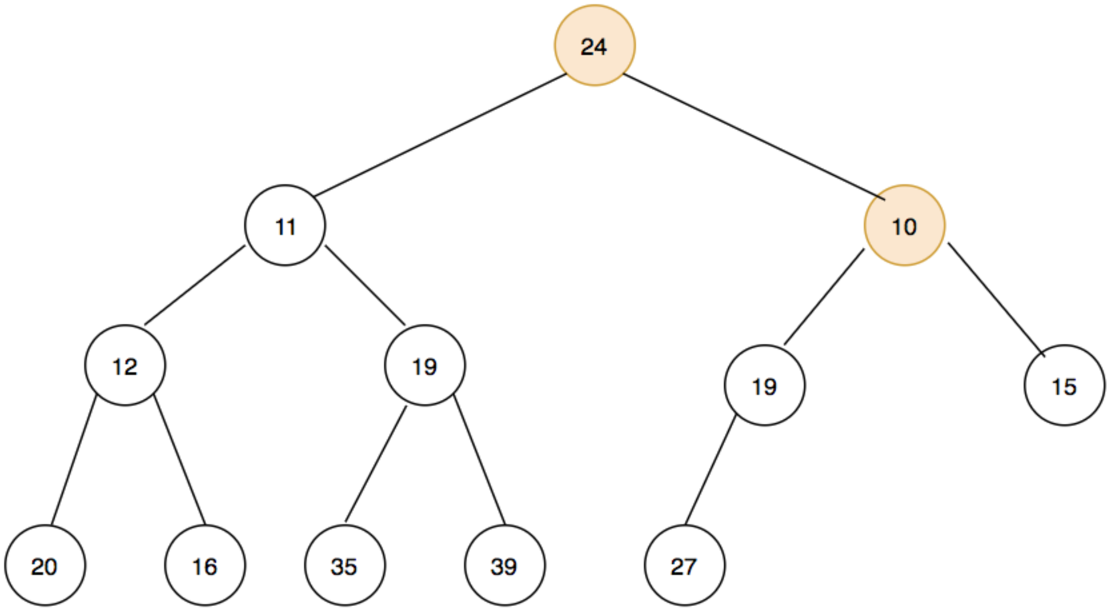
removeMin: part 2

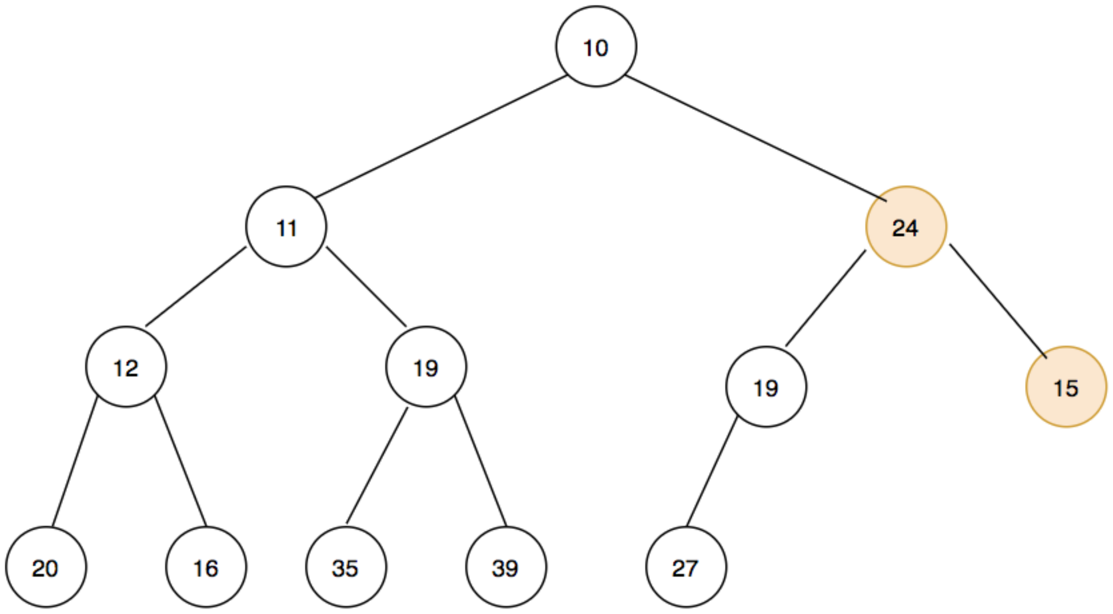


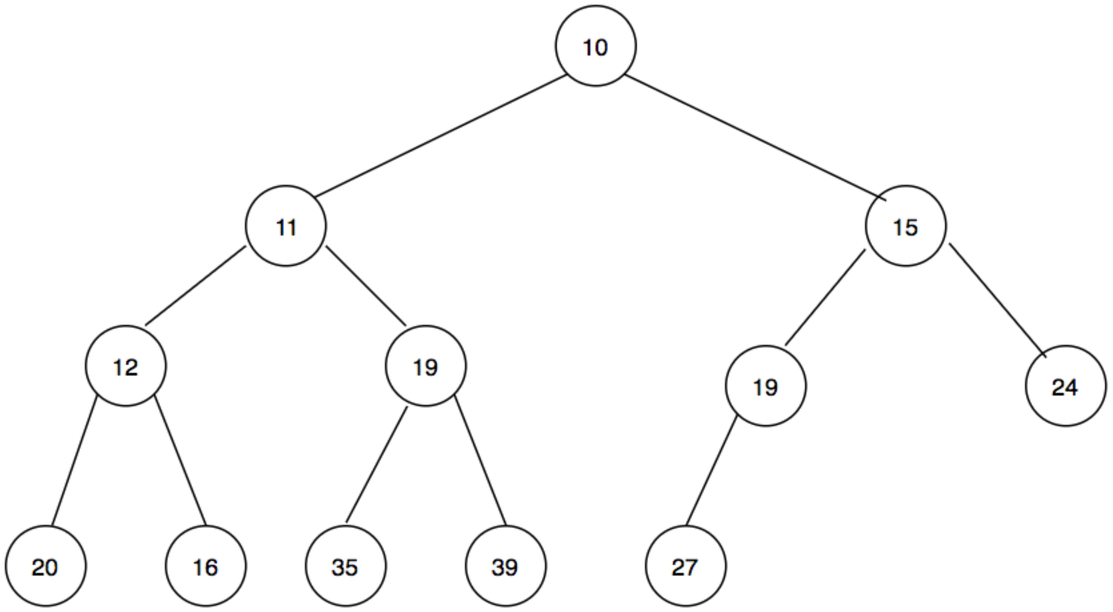




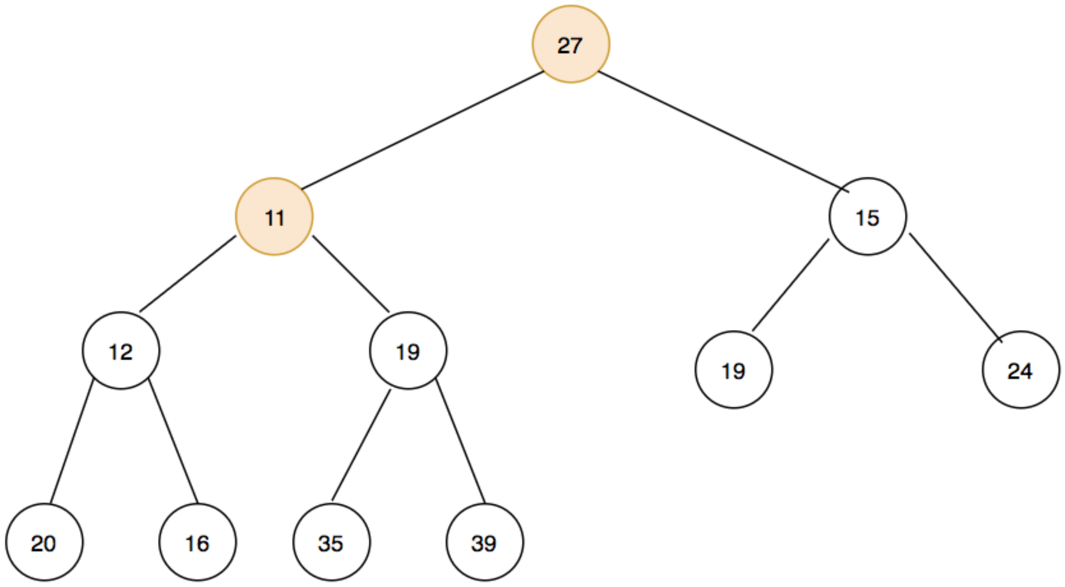
removeMin: part 3

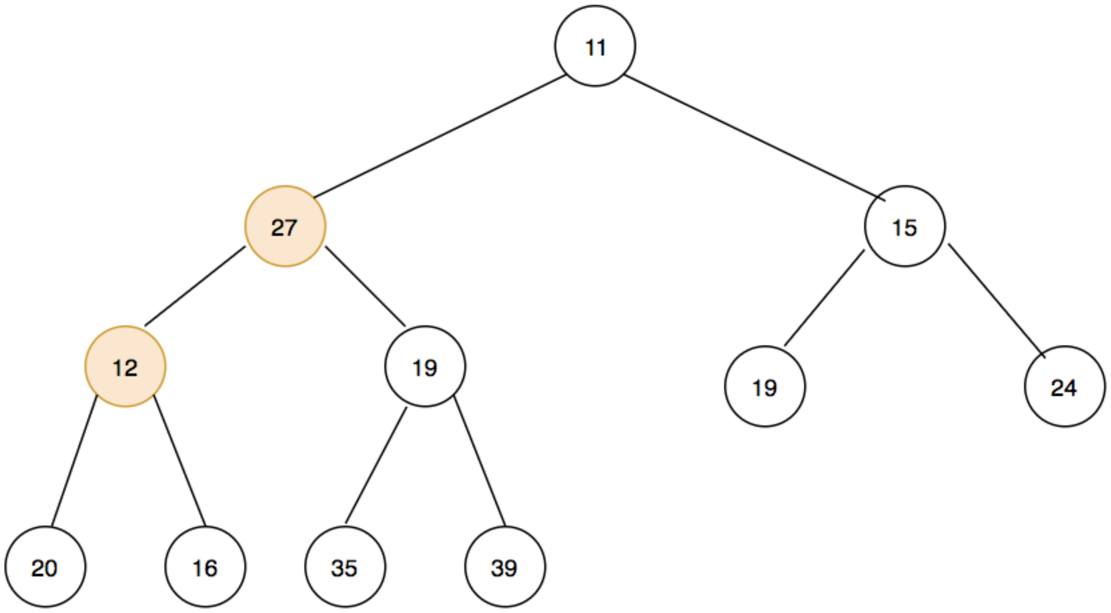


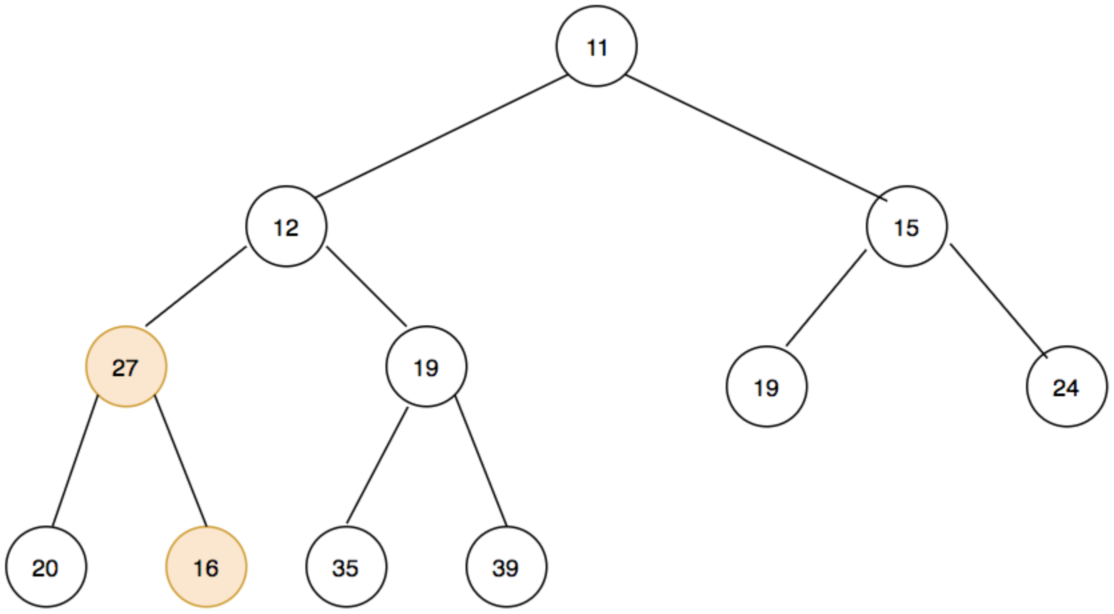

10
11
24
12
19
19
15
20
16
35
39
27

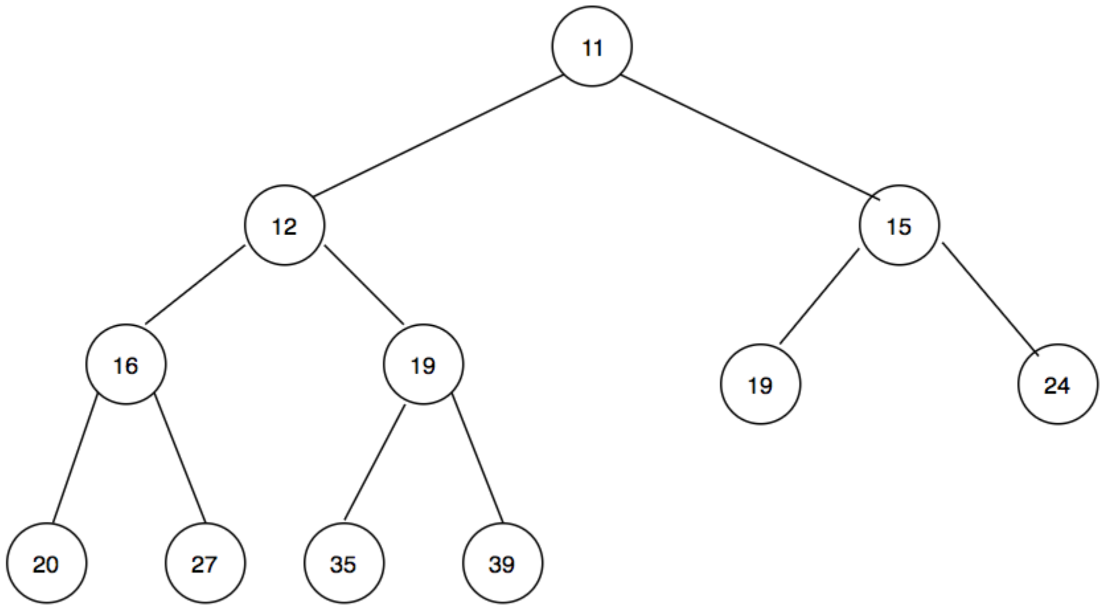

10
11
15
12
19
19
24
20
16
35
39
27

removeMin: part 4

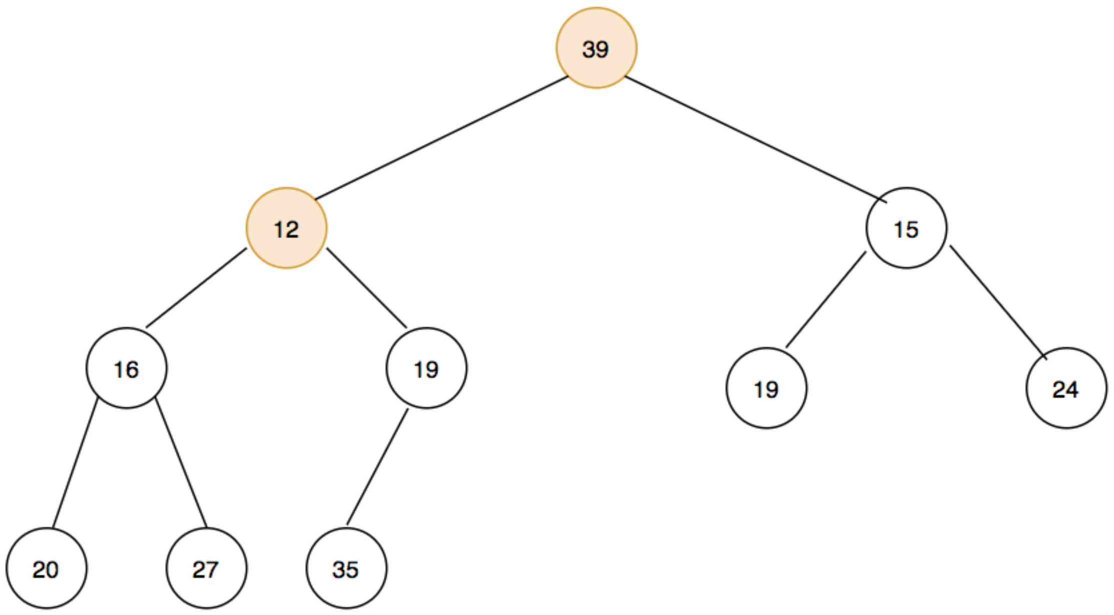


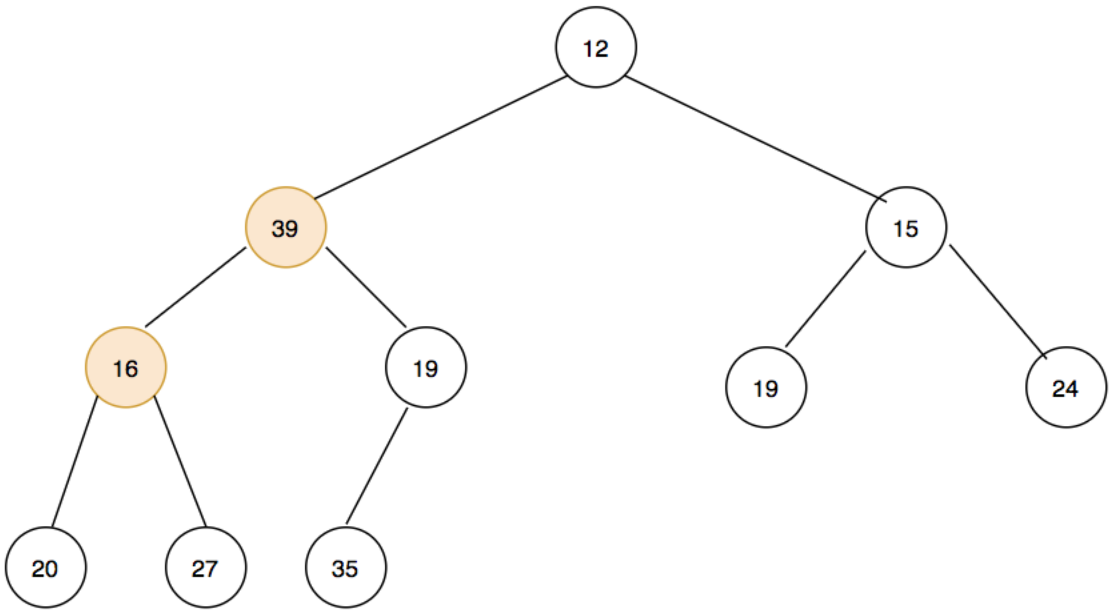


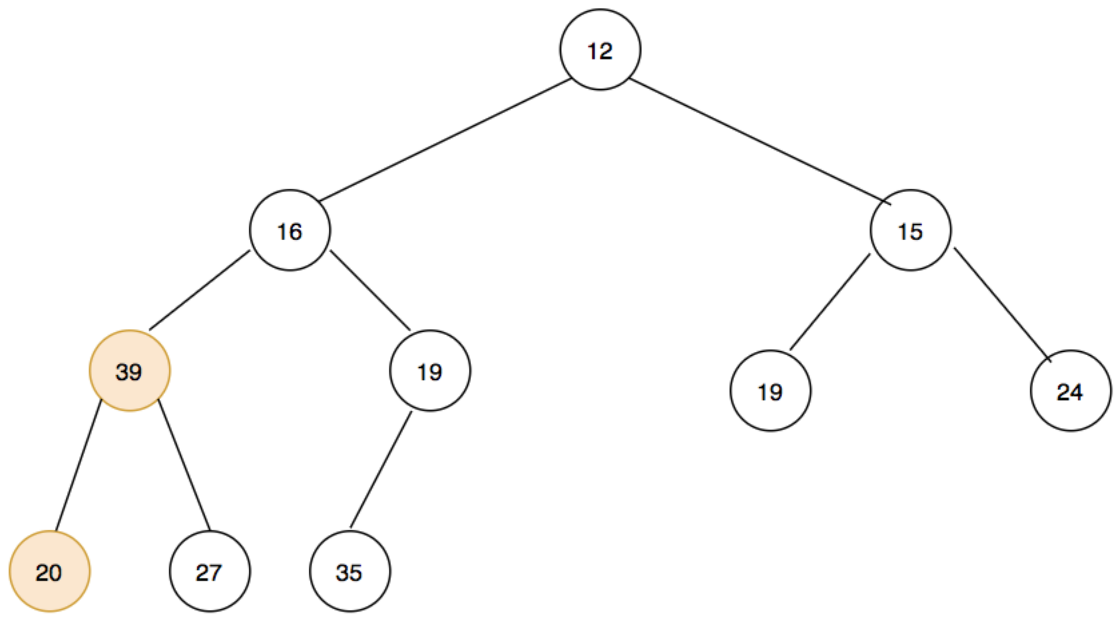
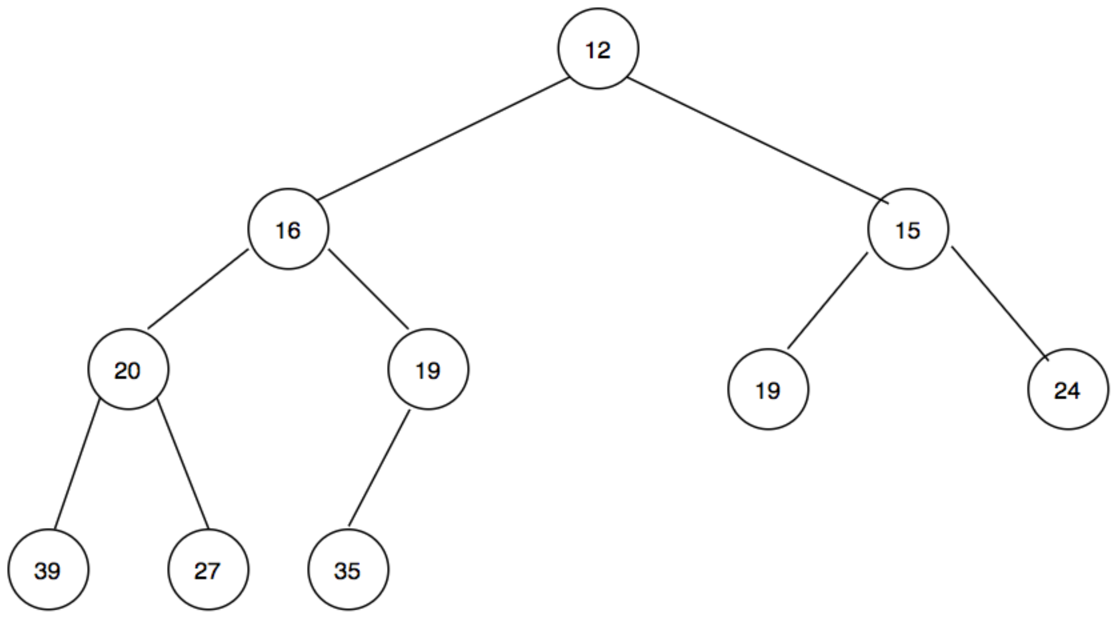




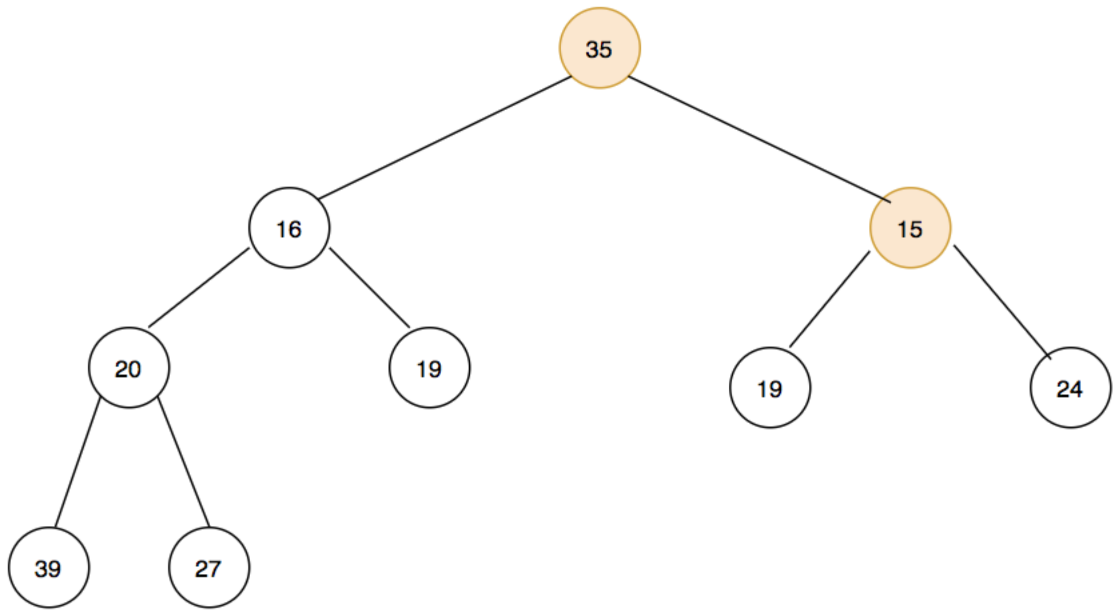
removeMin: part 5

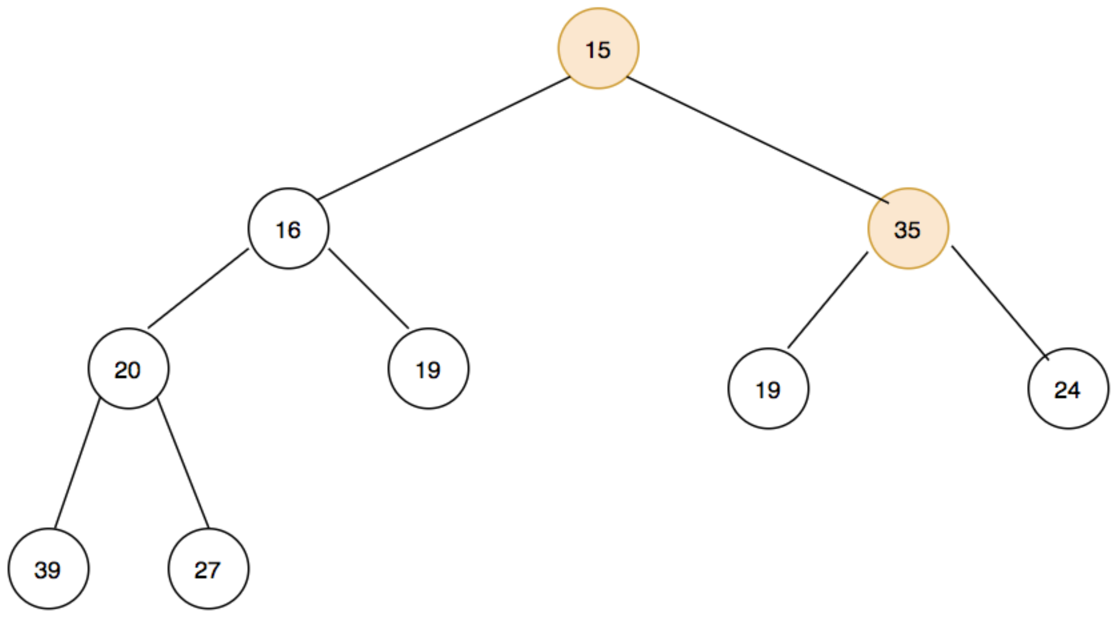

39
12
15
16
19
19
24
20
27
35

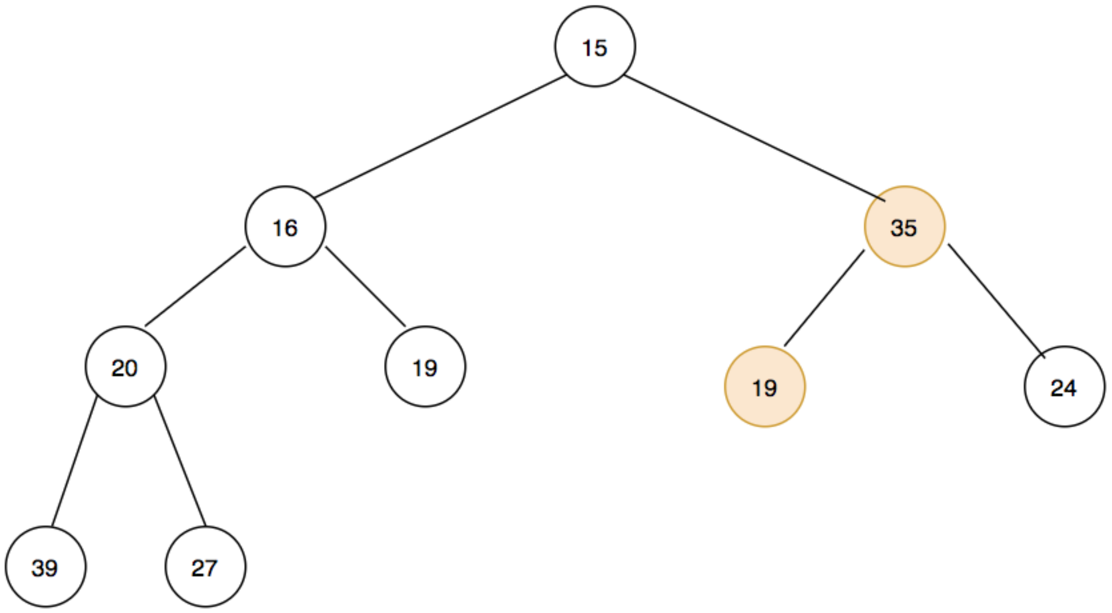

12
39
15
16
19
19
24
20
27
35

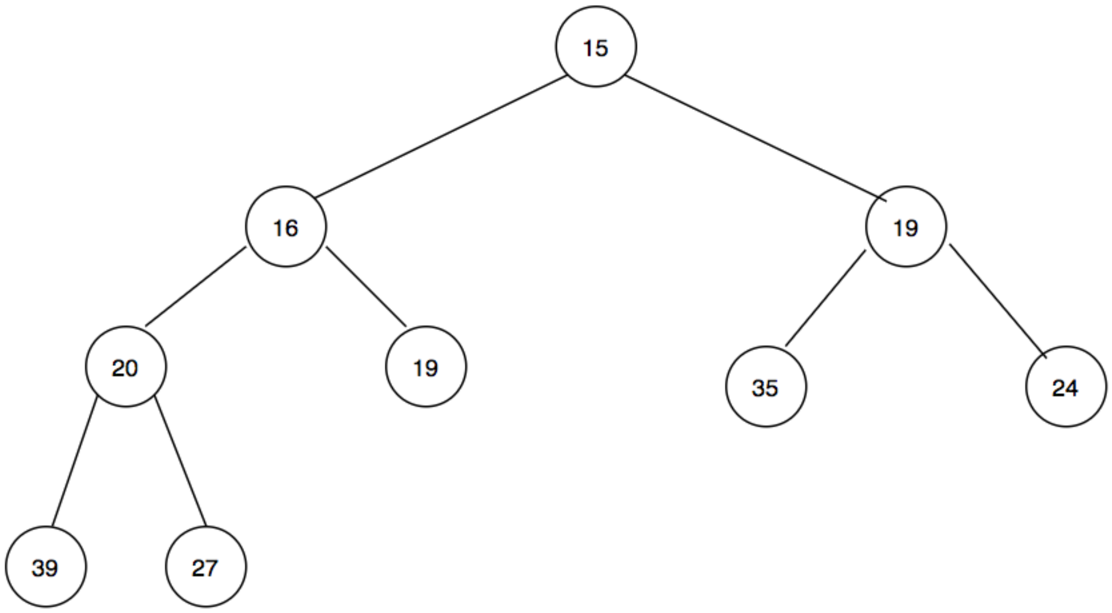

12
16
15
39
19
19
24
20
27
35

12
16
15
20
19
19
24
39
27
35

removeMin: part 6





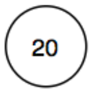

15
16
35
20
19
19
24
39
27


15
16
19
20
19
35
24
39
27

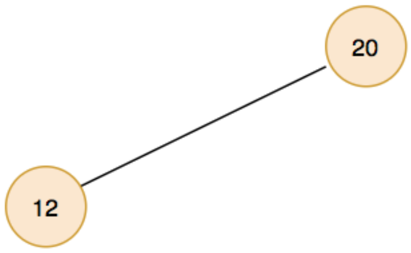
Question 2

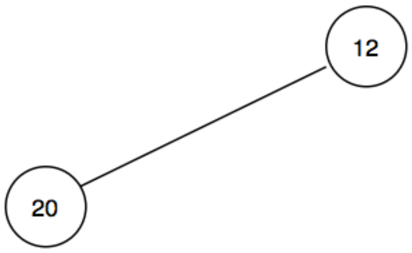
b)

insert 20

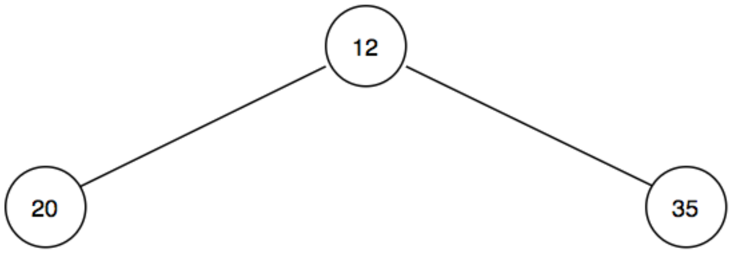


insert 12

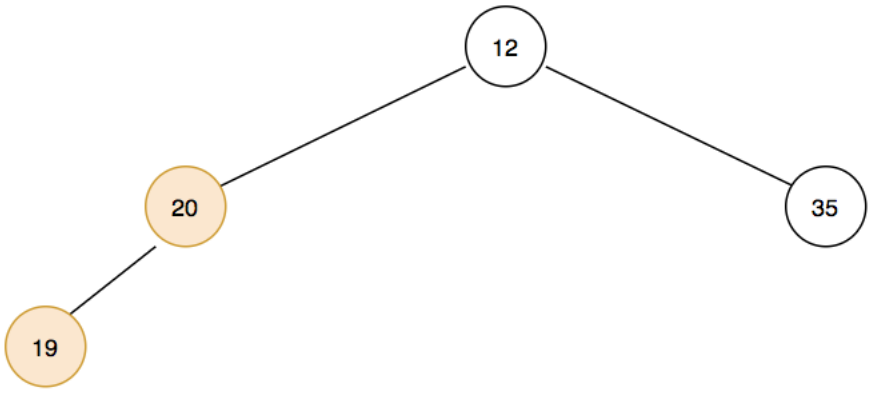


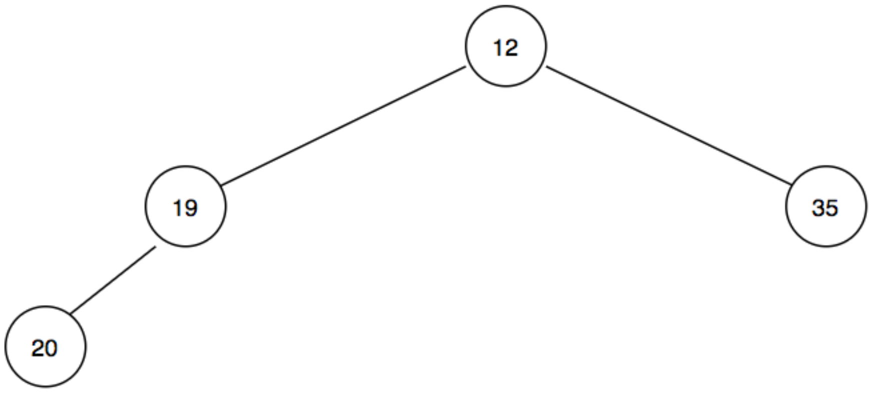


insert 35

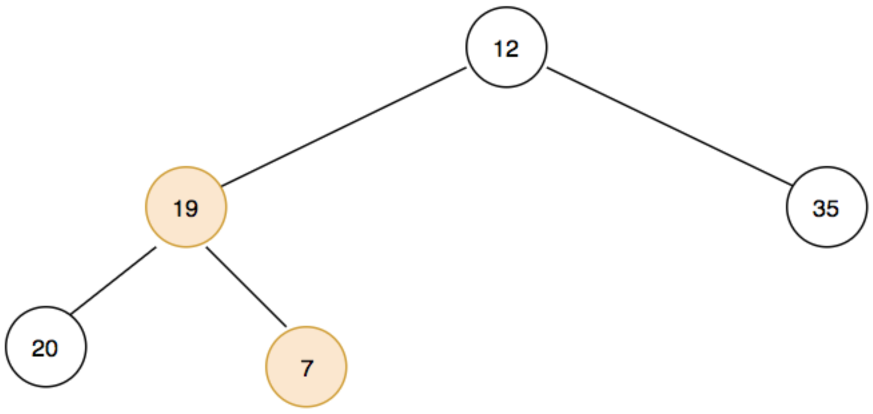


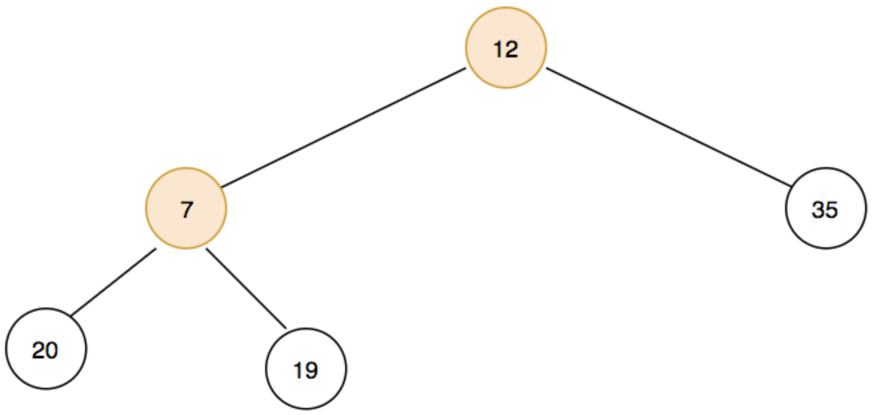
insert 19

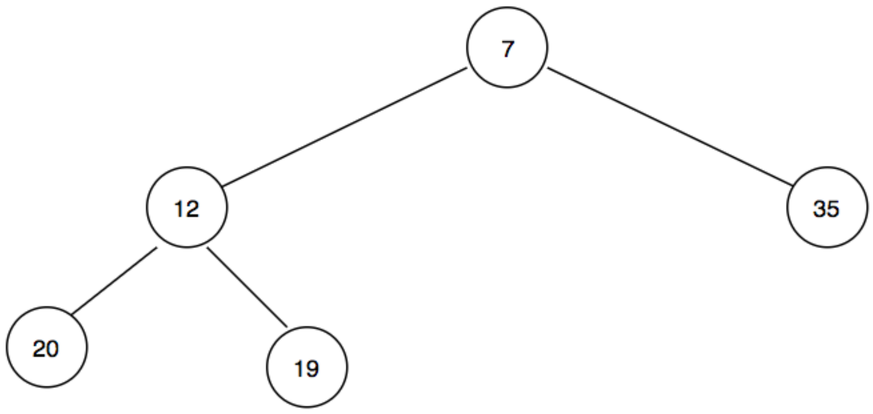




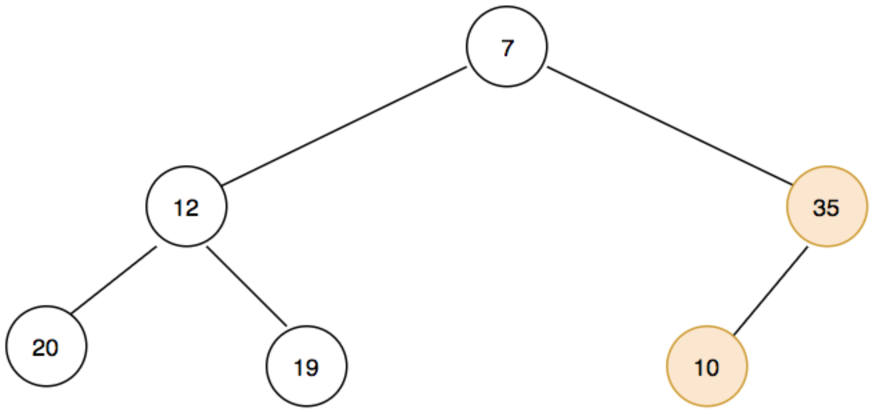
insert 7

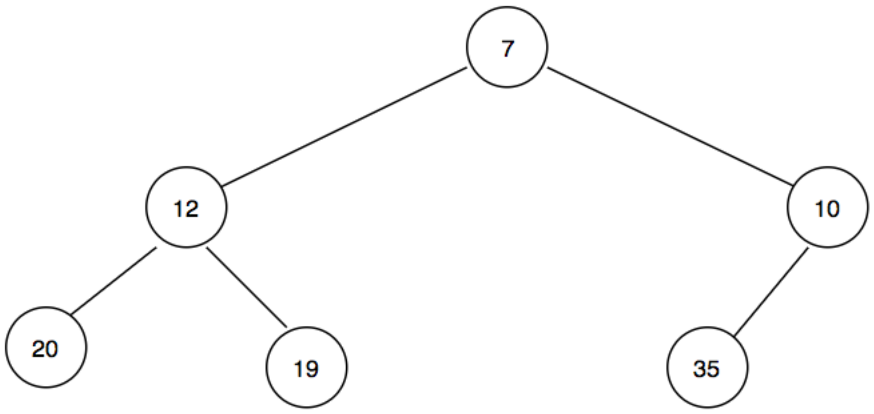

12
19
35
20
7


12
7
35
20
19

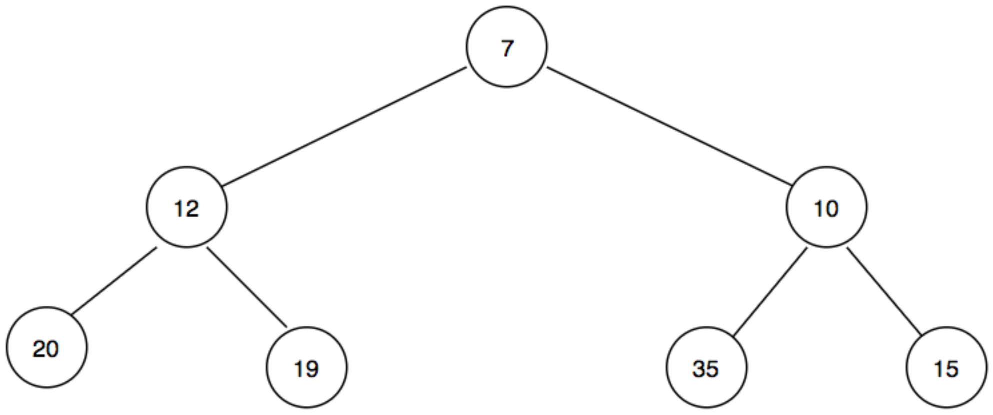

7
12
35
20
19

insert 10

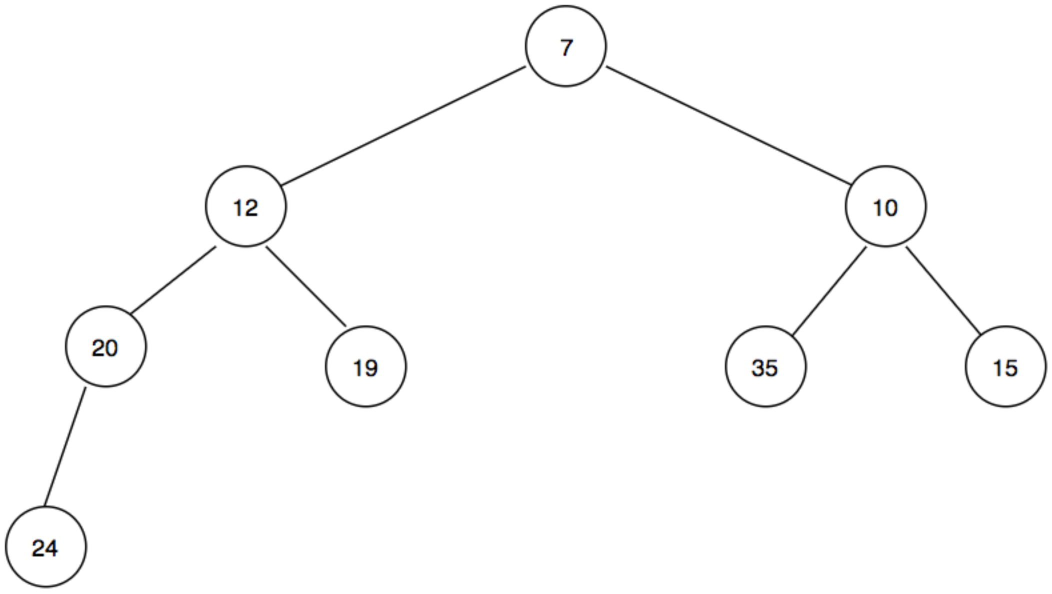




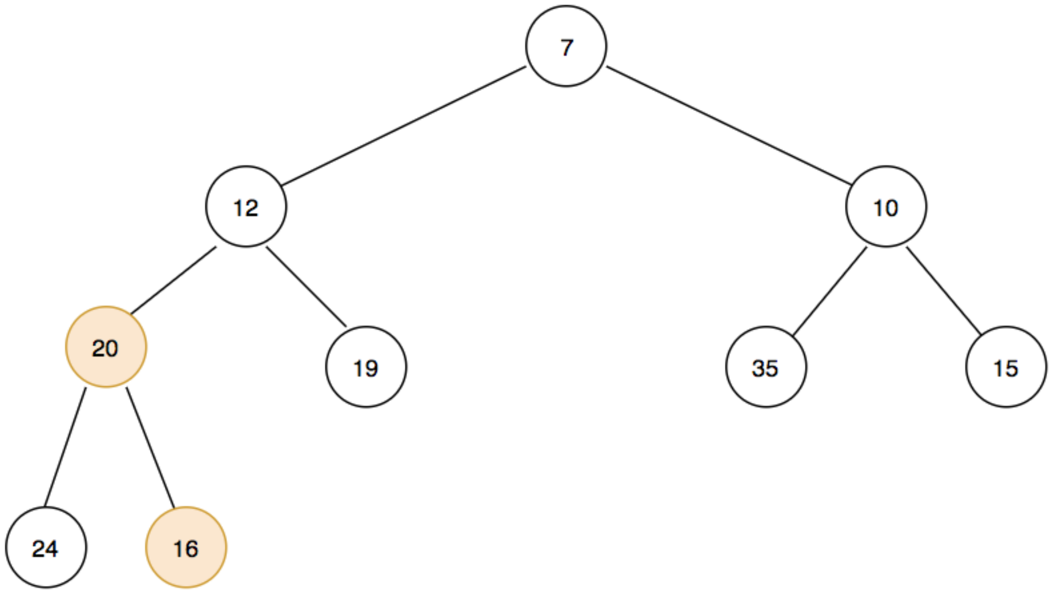
insert 15

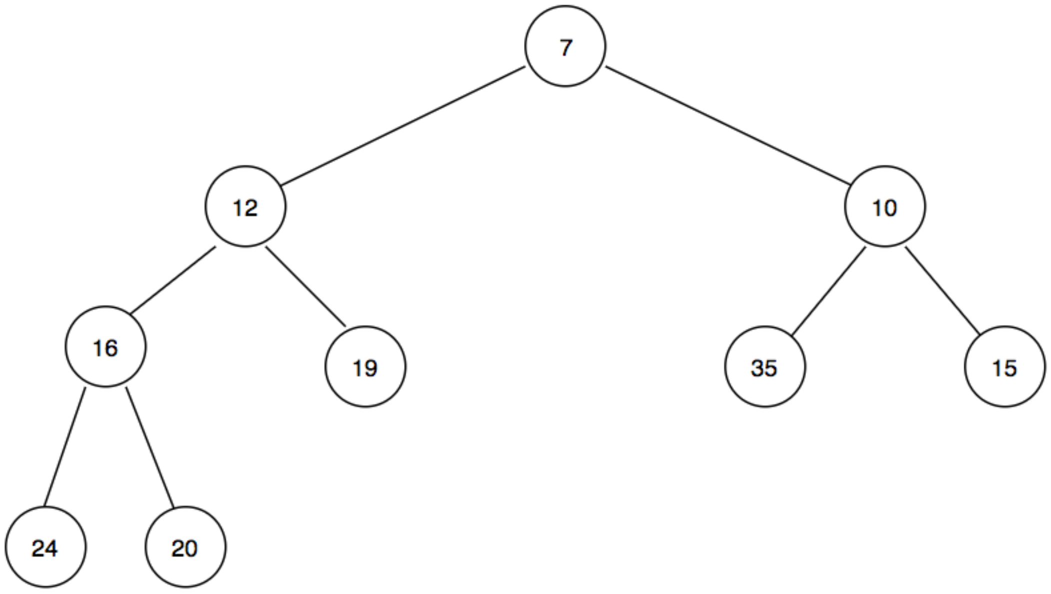


insert 24

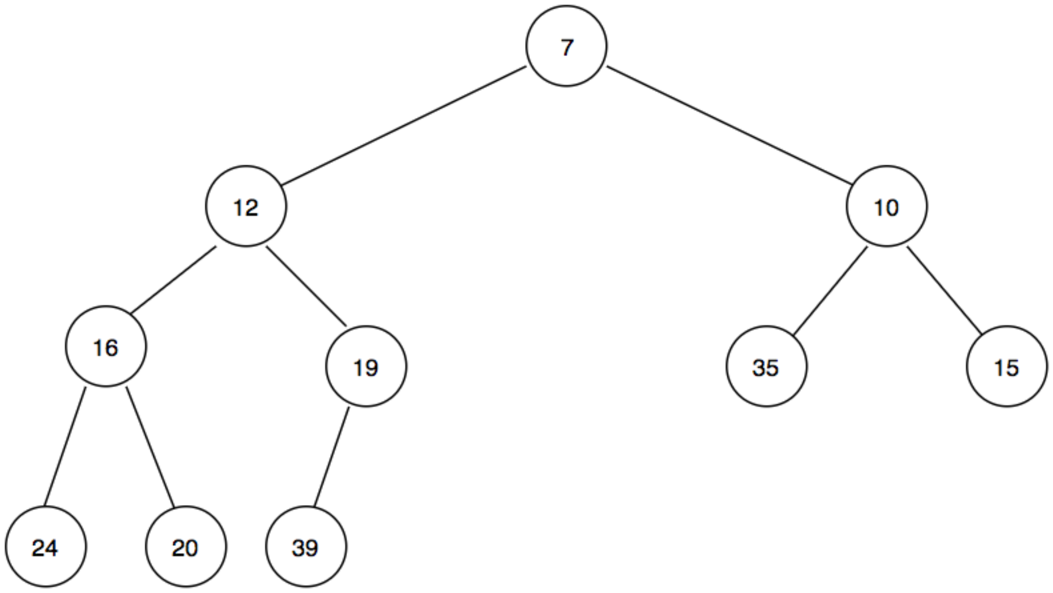


insert 16

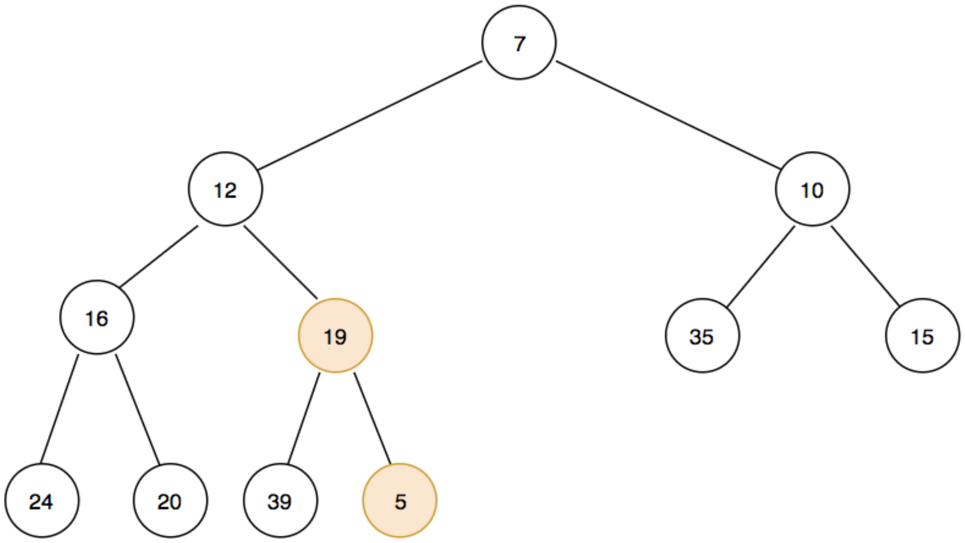


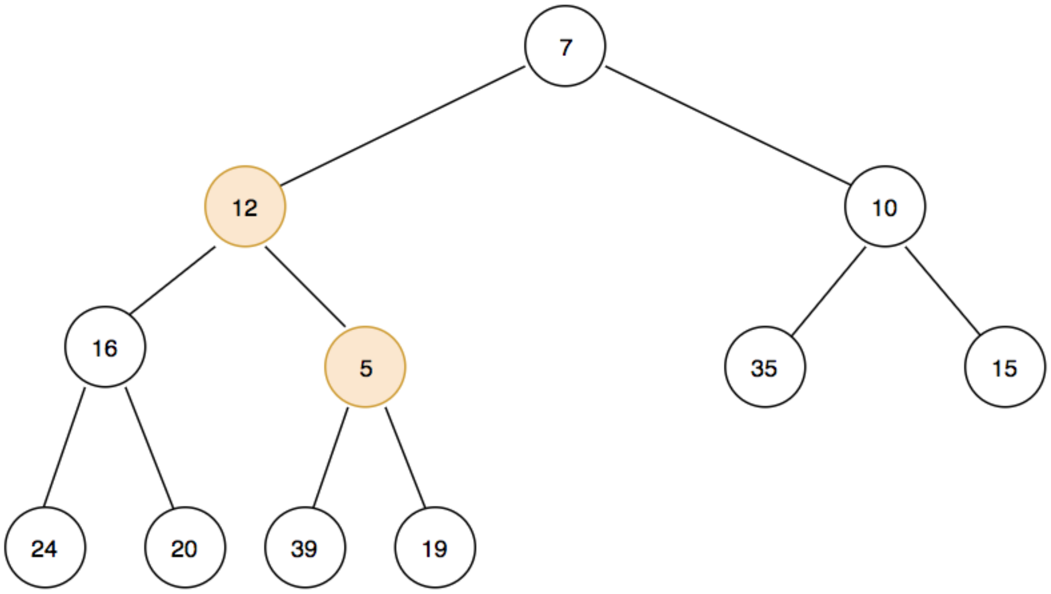


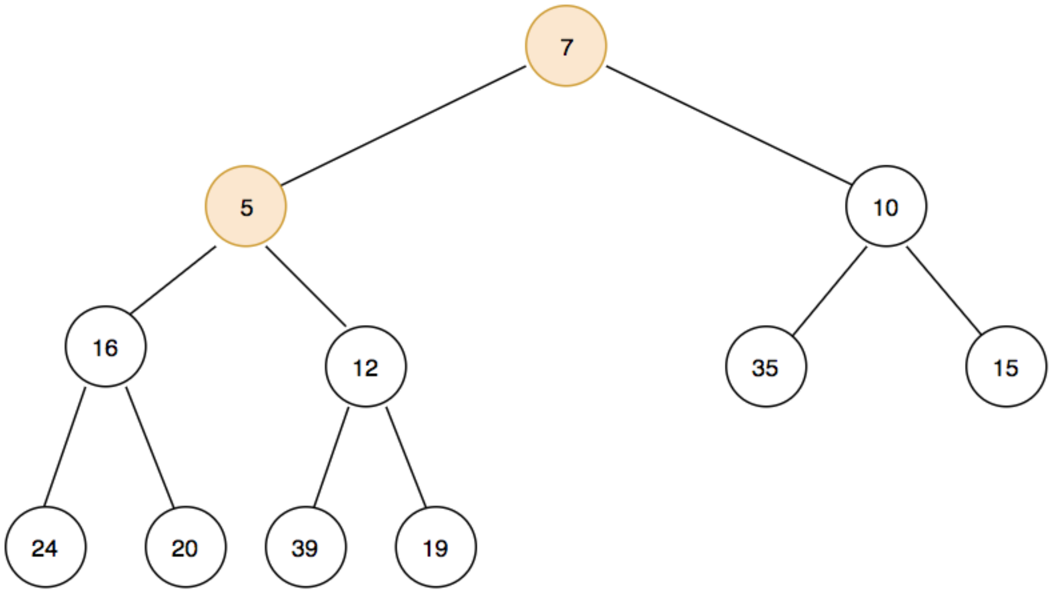
insert 39

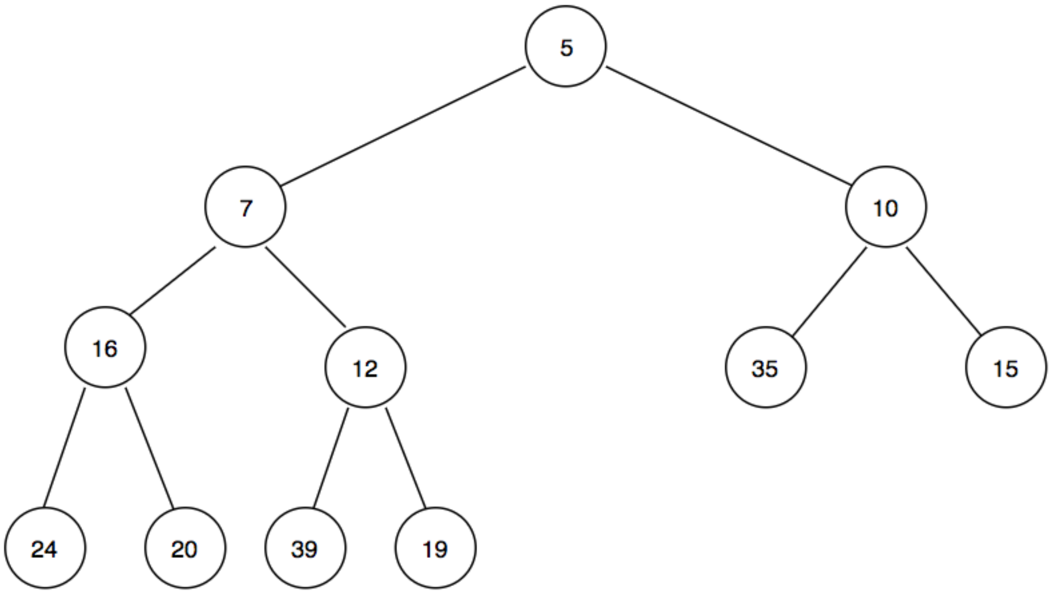


insert 5

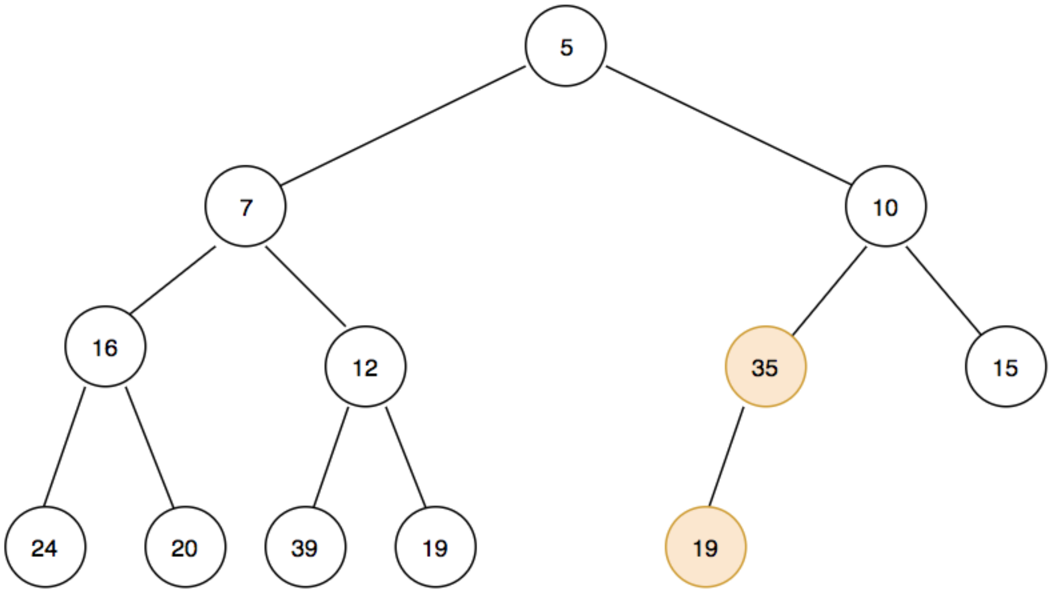


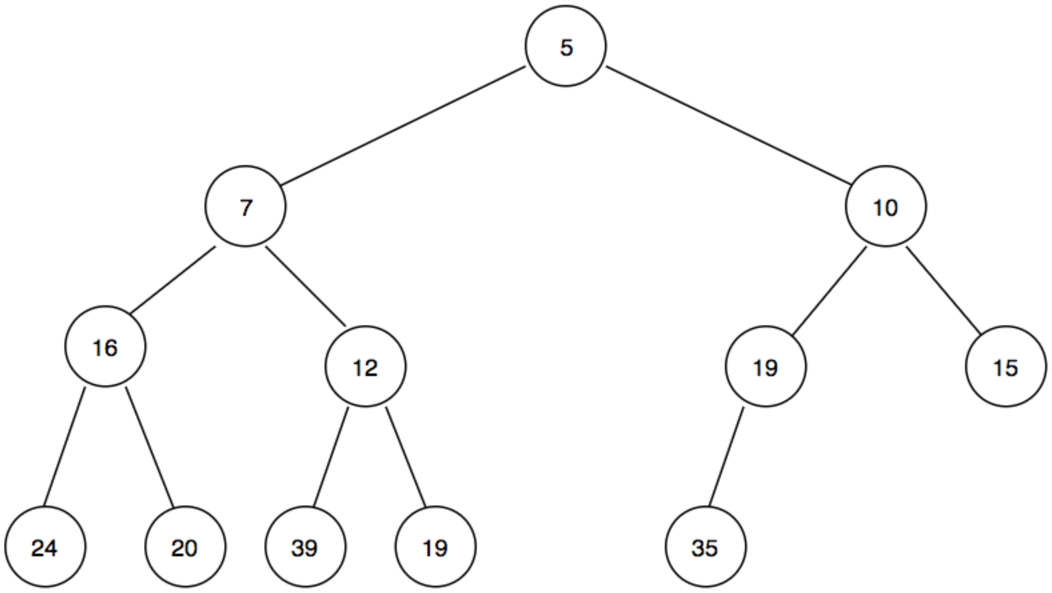

7
12
10
16
5
35
15
24
20
39
19


7
5
10
16
12
35
15
24
20
39
19

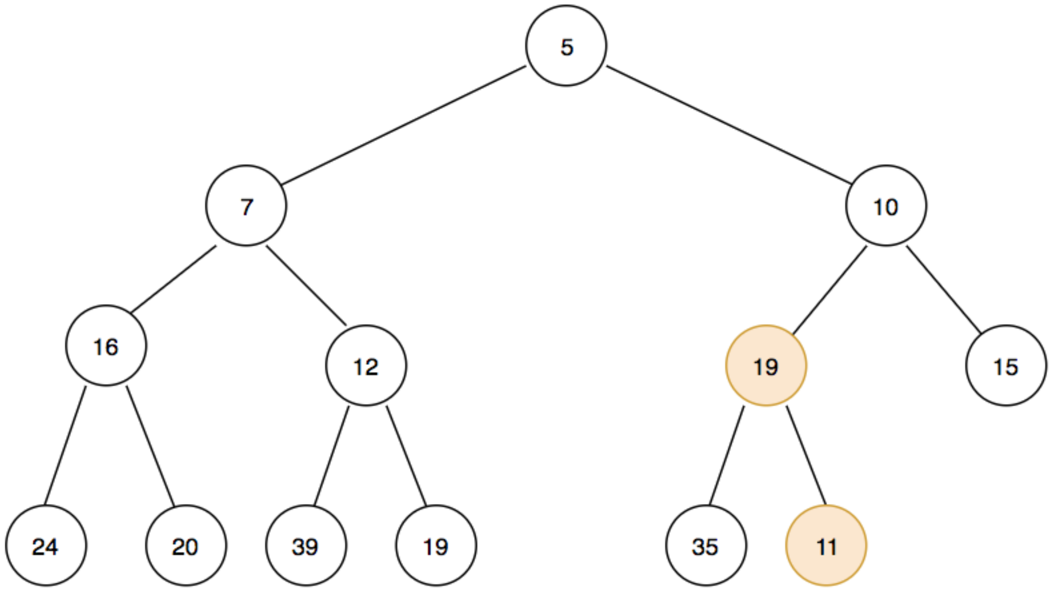


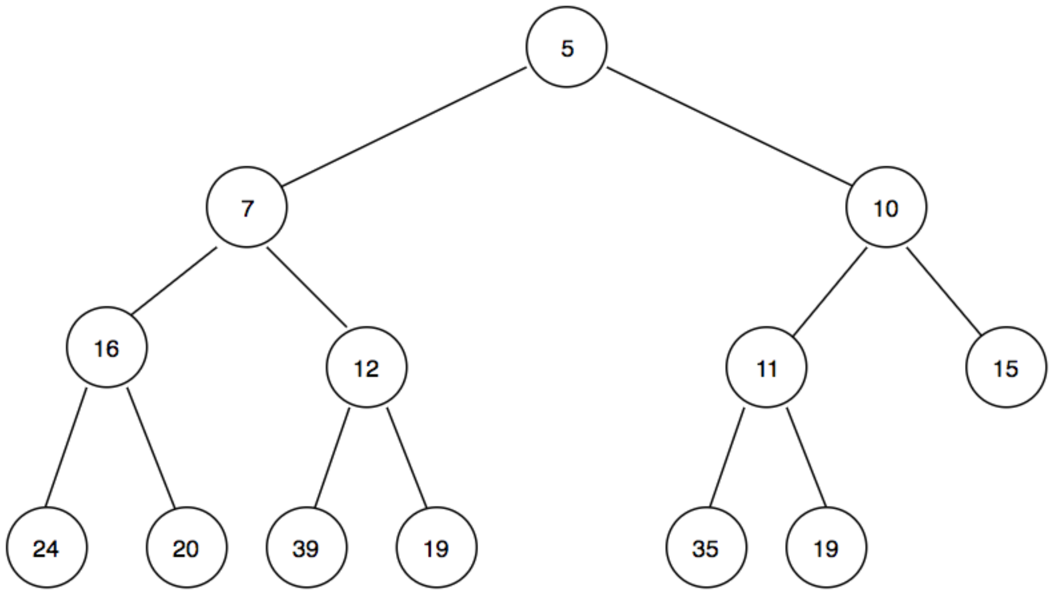
insert 19



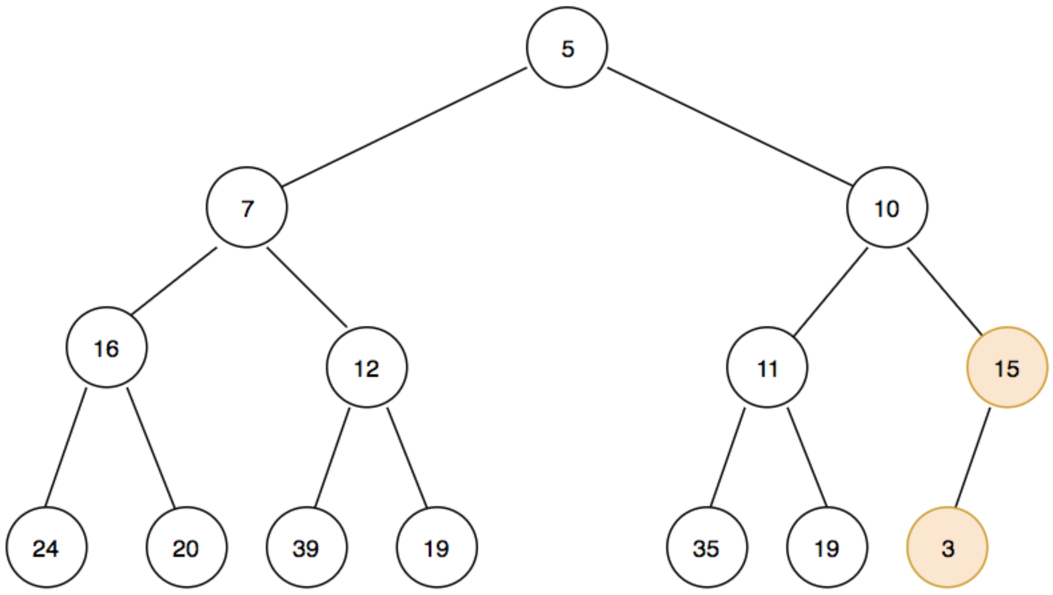


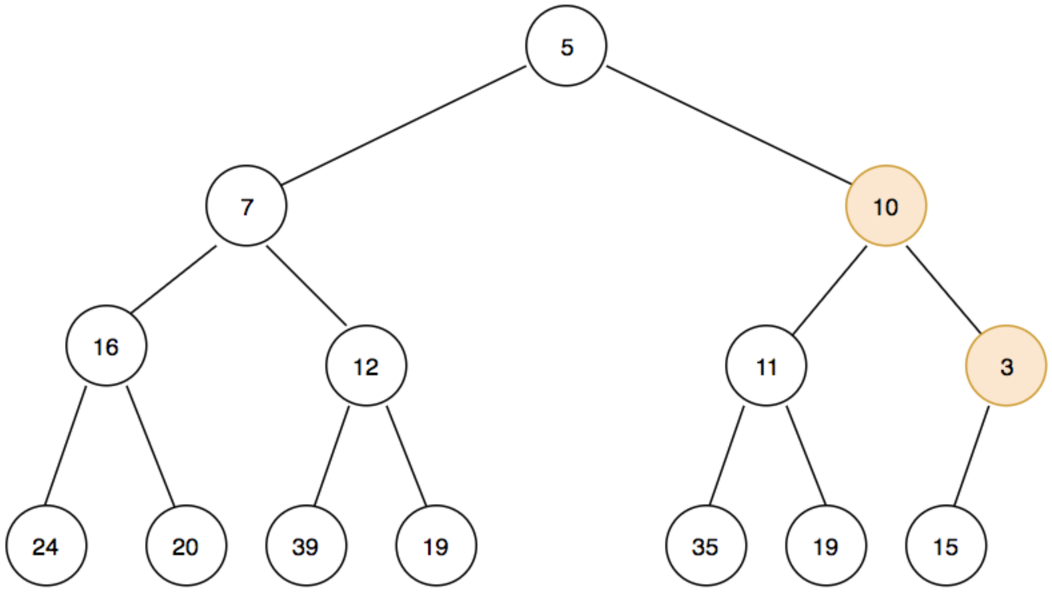
insert 11

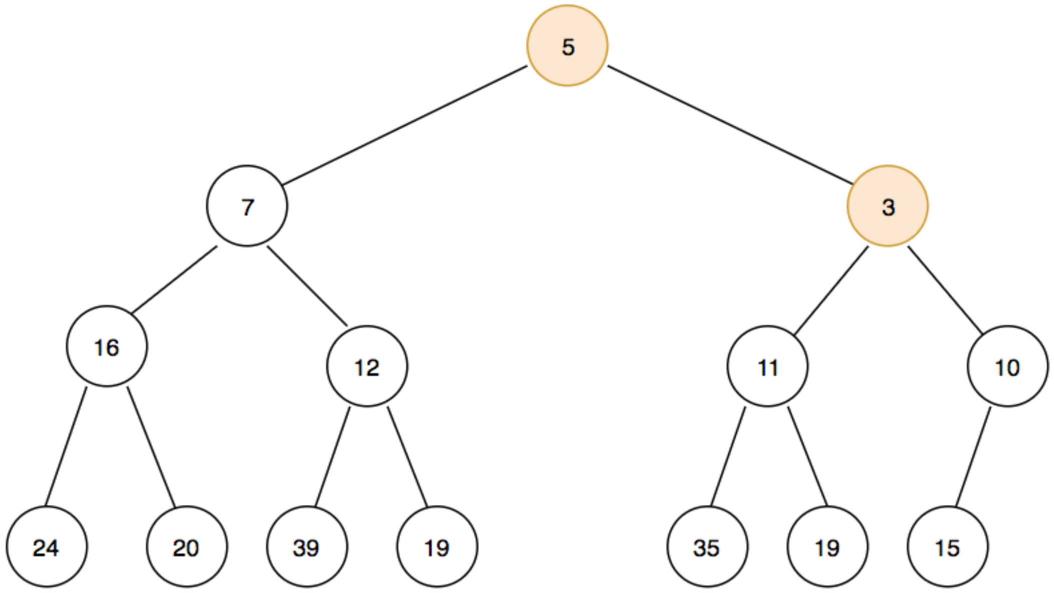


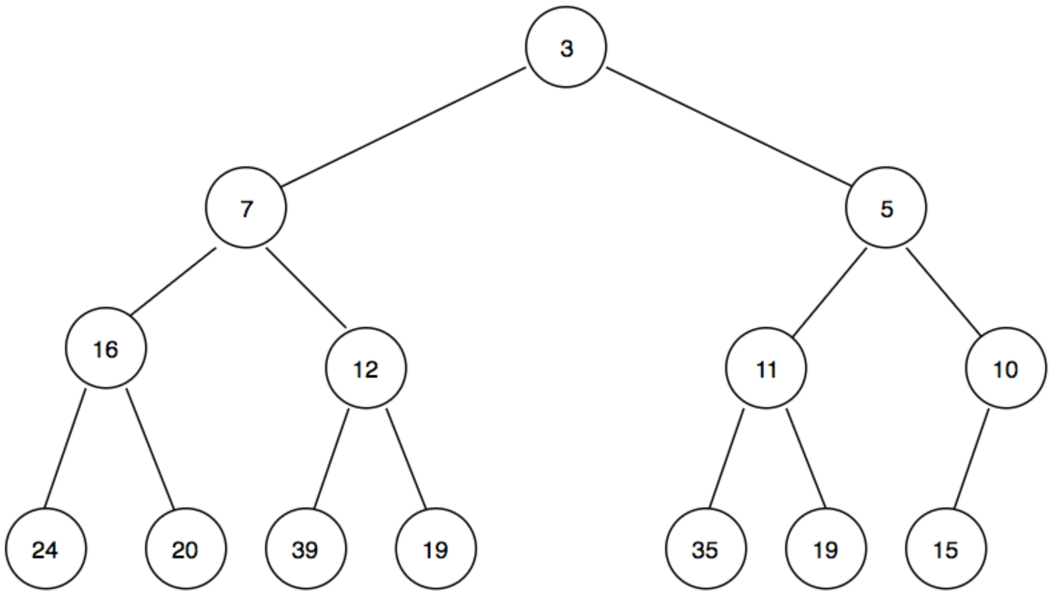


insert 3

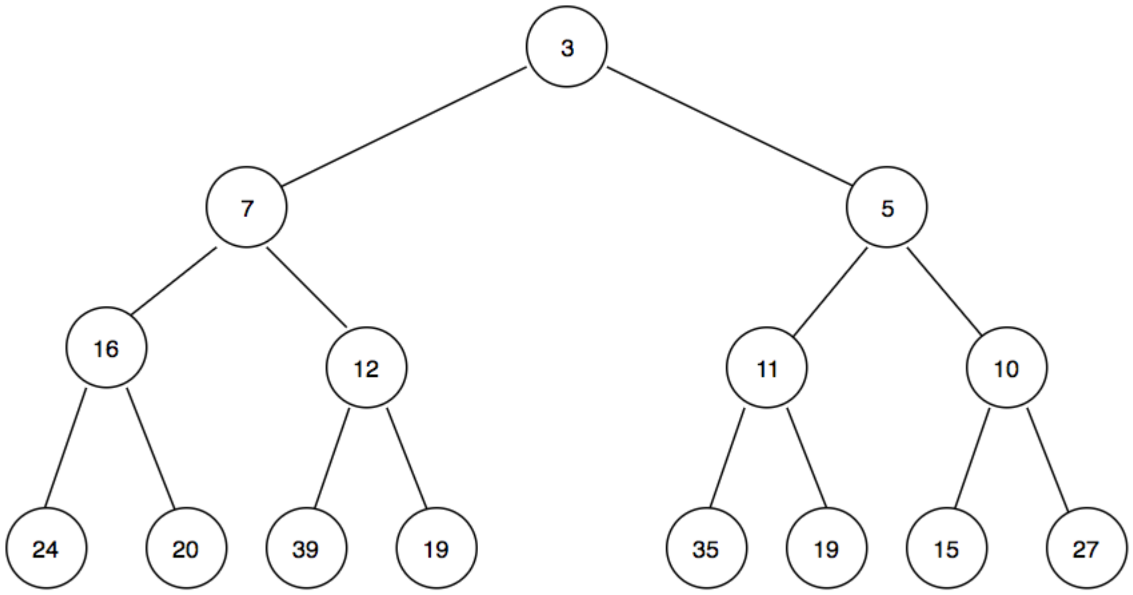



5
7
10
16
12
11
3
24
20
39
19
35
19
15


5
7
3
16
12
11
10
24
20
39
19
35
19
15

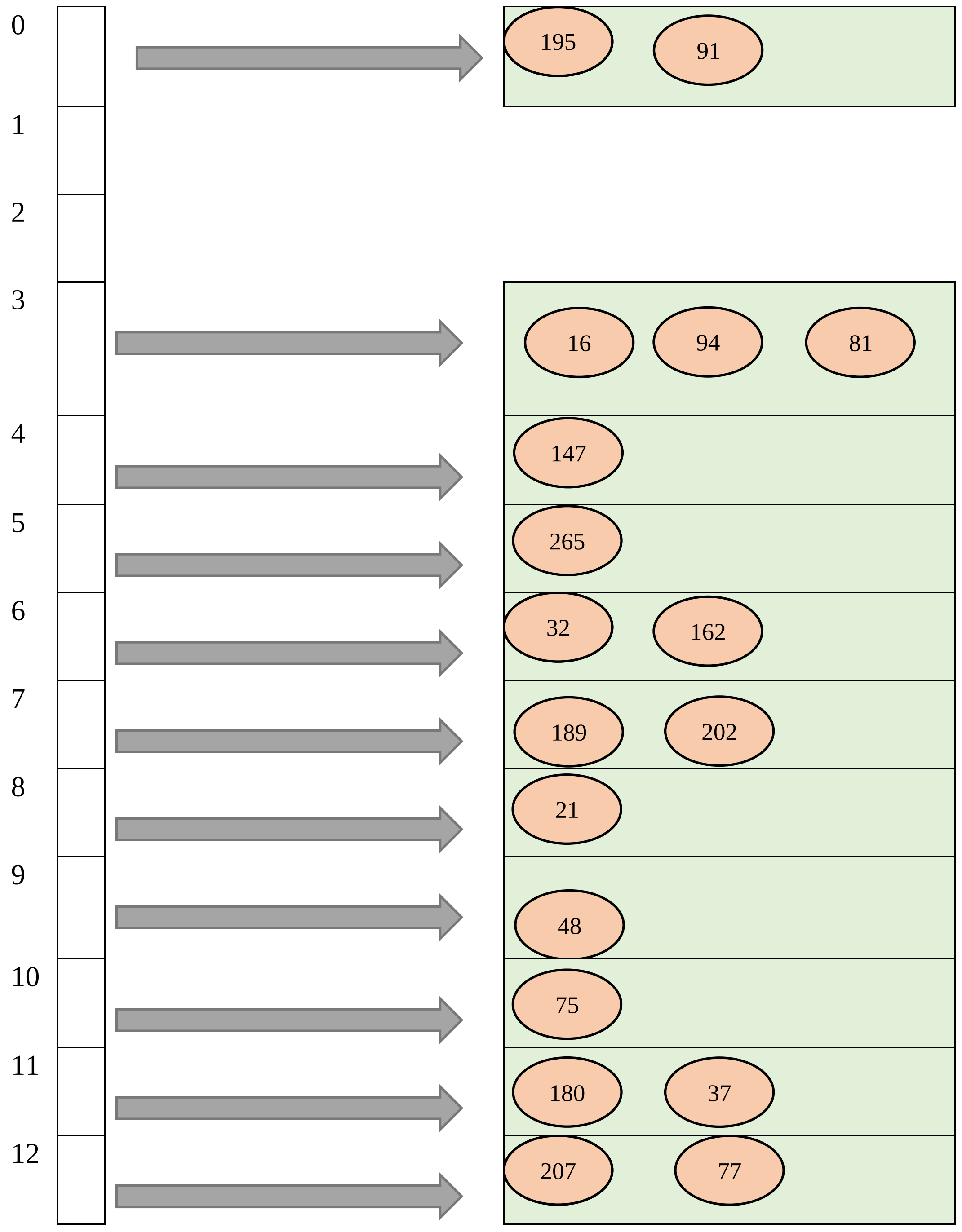

3
7
5
16
12
11
10
24
20
39
19
35
19
15

insert 27


3
7
5
16
12
11
10
24
20
39
19
35
19
15
27

Question 3

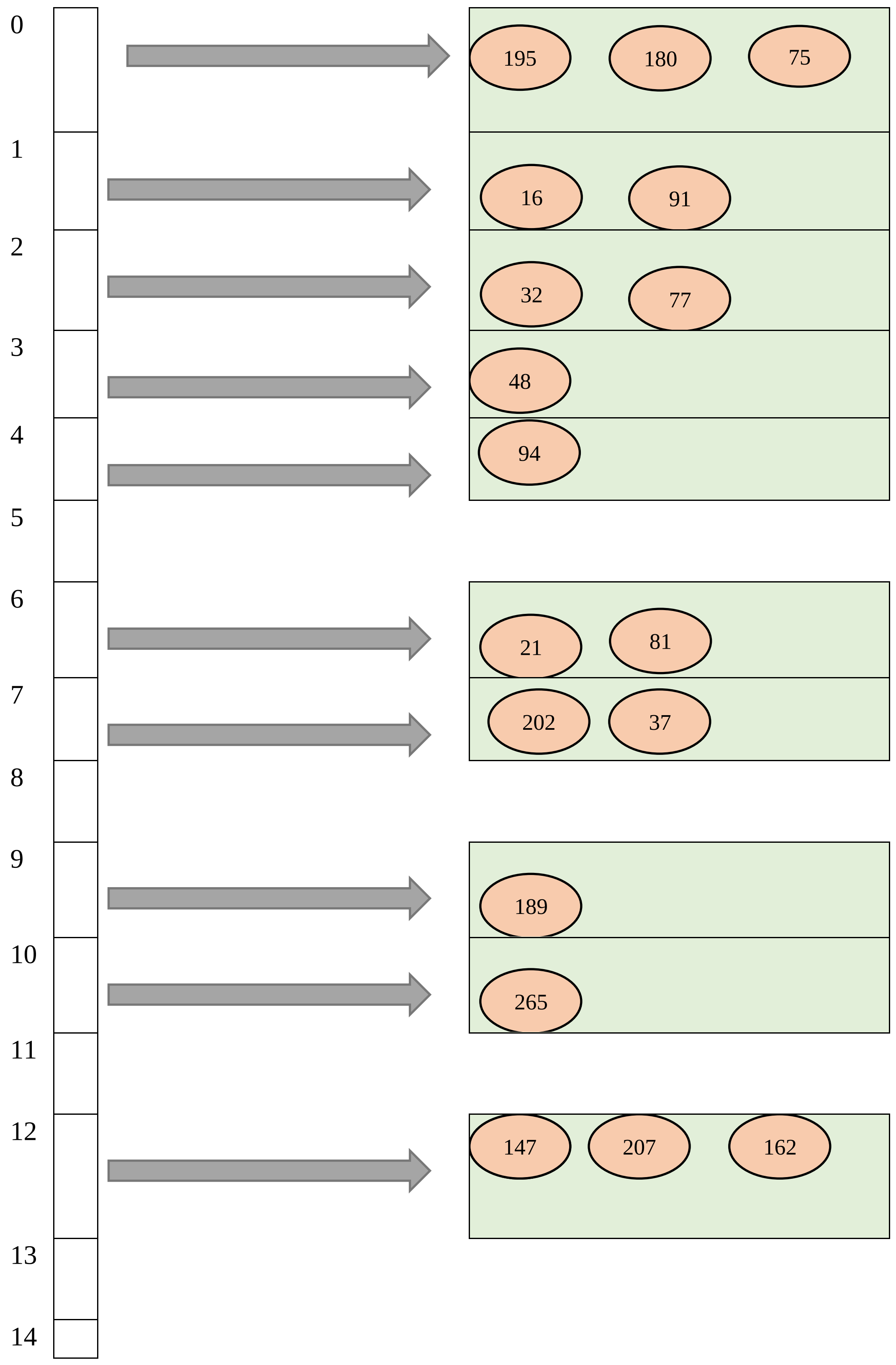
a)



b)

Collisions occur on the indices 0,3,6,7,11,12. Collisions occur when there is already a key present and we try to insert another one. Hence, we have 1+2+1+1+1+1= 7 max collisions.

Question 4


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
195
180
75
16
91
32
77
48
94
21
81
202
37
189
265
147
207
162

The load factor is defined as the number of keys stored in the hash table divided by the capacity. The number of keys is in this new hash table is held constant, but the capacity is increased by 15.38%. The load factor is smaller. The greater the capacity, the less occupied the hash table might be, and hence the less the probability of collisions occurring. For this particular example, the number of collisions actually increases! Number of collisions is: 2+1+1+1+1+2=8>7.
The proposal does hold validity to it, but for this particular example, it might be better to increase the array even more. In the first example, the load factor is = 18/13 and in the second example, 18/15. For separate chaining it is advised that the load factor be less than 0.9. So, perhaps the array size should be at least 20. Increasing it to 15 will not have a big impact.
Also be careful, if the load factor approaches 0, then the separate chaining may actually be wasteful in terms of space.

<u>Question 5</u>

i)

k=25
h(k)=6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
						25												

k=12
h(k)=12

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
						25						12						

k=42
h(k)=4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
				42		25						12						

k=31
h(k)=12
d(k)=4
Probe: 12, 16
Collisions: 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
				42		25						12				31		

k=35
h(k)=16
d(k)=7
Probe: 16, 4, 11
Collisions: 1+2=3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
				42		25					35	12				31		

k=39
h(k)=1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	39			42		25					35	12				31		

Remove (31)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	39			42		25					35	12						

k=48
h(k)=10

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	39			42		25				48	35	12						

Remove (25)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	39			42						48	35	12						

k=18
h(k)=18

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	39			42						48	35	12						18

k=29
h(k)=10
d(k)=6
Probe: 10,16
Collisions: 3+1=4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	39			42						48	35	12				29		18

k=29
h(k)=10
d(k)=6
Probe: 10,16,3
Collisions: 4+2=6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	39		29	42						48	35	12				29		18

k=35
h(k)=16
d(k)=7

Probe: 16, 4, 11, 18, 6
Collisions: 6+4=10

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	39		29	42		35				48	35	12				29		18

ii) The longest cluster is of size 3. This occurs at indices 10,11,12. This cluster is obtained after we put 12,35, then 48.

iii) We have 10 total collisions as a result of the above insertions. I have indicated the number of cumulative collisions above, for every step in which the collisions increase. Interestingly, the last step yields the highest number of collisions (4).

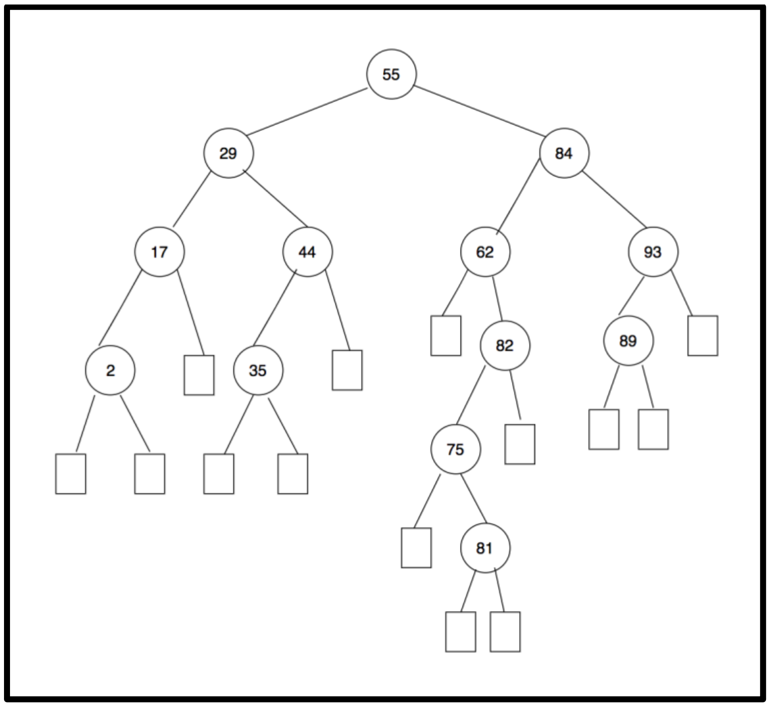
iv) The load factor is defined as the number of keys stored in the hash table divided by the capacity. At the end, it is 9/19=0.474.

Question 6

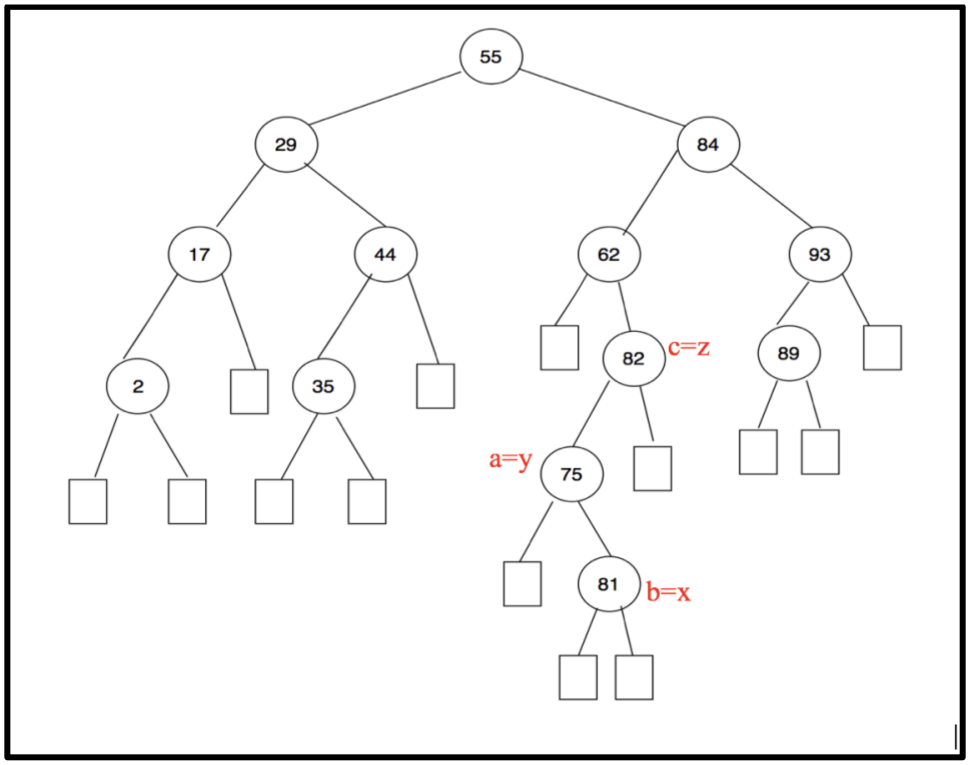
i) Yes, there are errors with the given AVL tree.

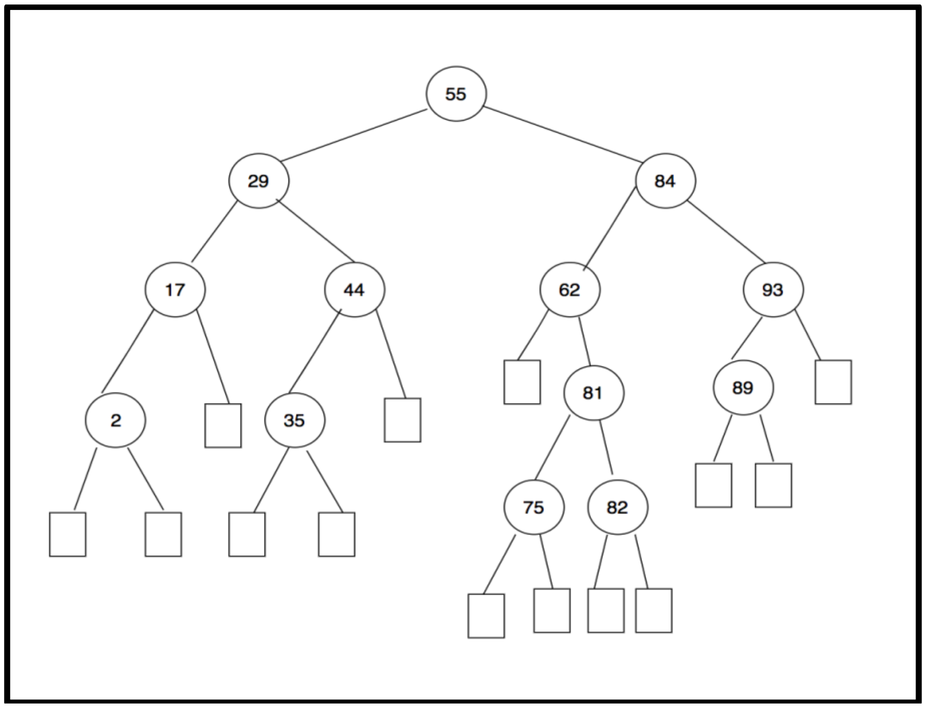
1) The location of 2 is incorrect. 2 is placed in the right subtree of 55, implying that it is larger than 55, which is wrong. It must be placed at the far left side of the tree.
2) The height balance property is not maintained. The tree rooted at 82 is not balanced because the left subtree has height 2 while the right has height 0. The tree rooted at 62 is not balanced because the left subtree has height 1 while the right subtree has height 3. The tree rooted at 84 is not balanced because the left subtree has height 4 and the right subtree has height 2. The tree rooted at 55 is not balanced because the left subtree has height 3 and the right subtree has height 5. To fix the AVL tree, I will not delete any items. I will simply change the location of 2. Then, restructure the subtree.

Step 1: Correct location of 2

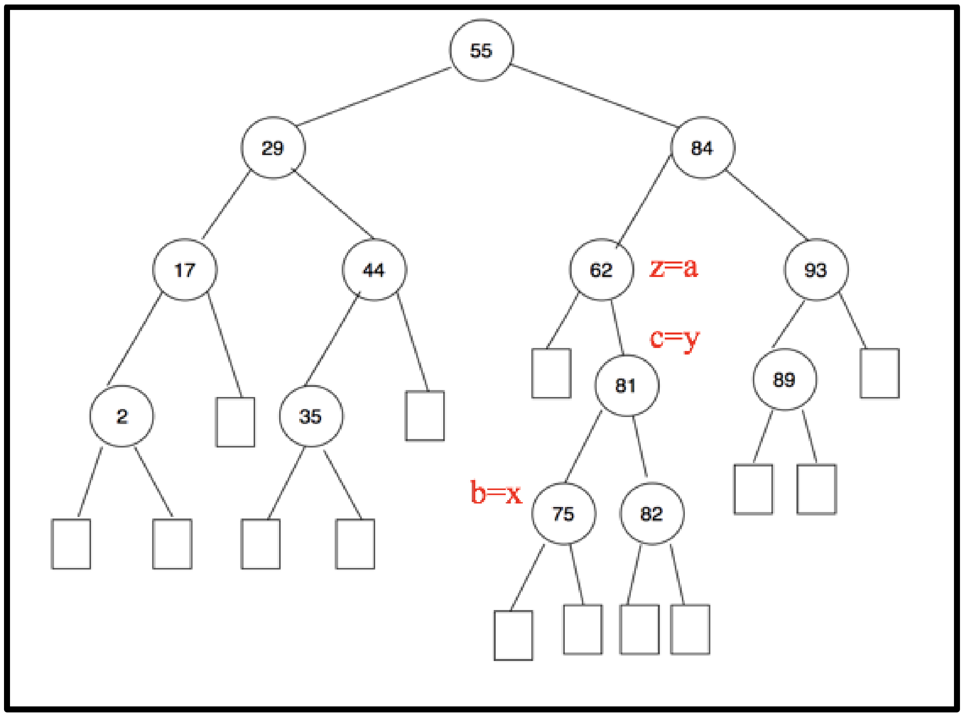


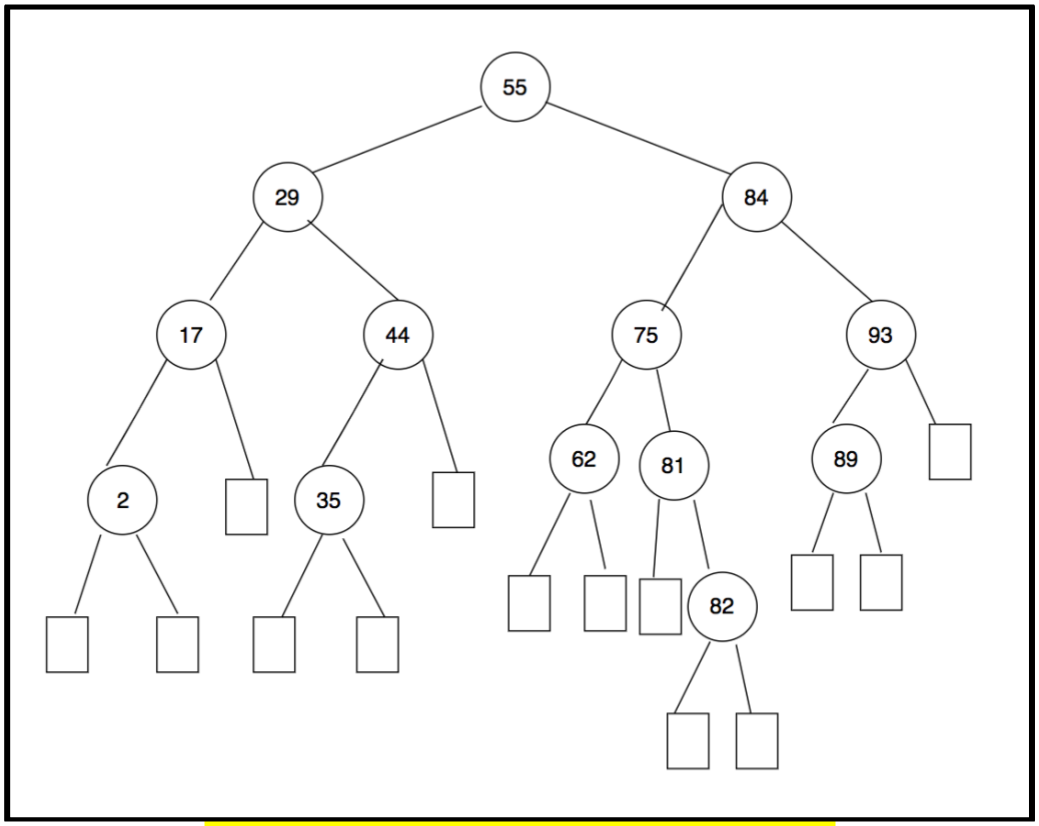
Step 2: Now, notice that the subtree rooted at 82 is not balanced. The left subtree has height 2 while the right subtree has height 0.
Z is the 1st encountered node causing unbalance (82). Y is the child of z with the higher height (75). X is the child of Y with the higher height (81). So set c=z (82), a=y (75), b=x (81), and apply a double rotation for this subtree. In other words replace z with b, take a and c and make them children of b. Maintain in-order traversal of subtrees.





Step 3: Now, notice that the subtree rooted at 62 is not balanced. The left subtree has height 0 while the right subtree has height 2. Here I can either perform a single rotation or a double rotation. I will choose the latter. Z is the 1st encountered node causing unbalance (62). Y is the child of z with the higher height (81). X is the child of Y with the higher height (75) (or 82)
I will set z=a (62), c=y (81), b=x (75).



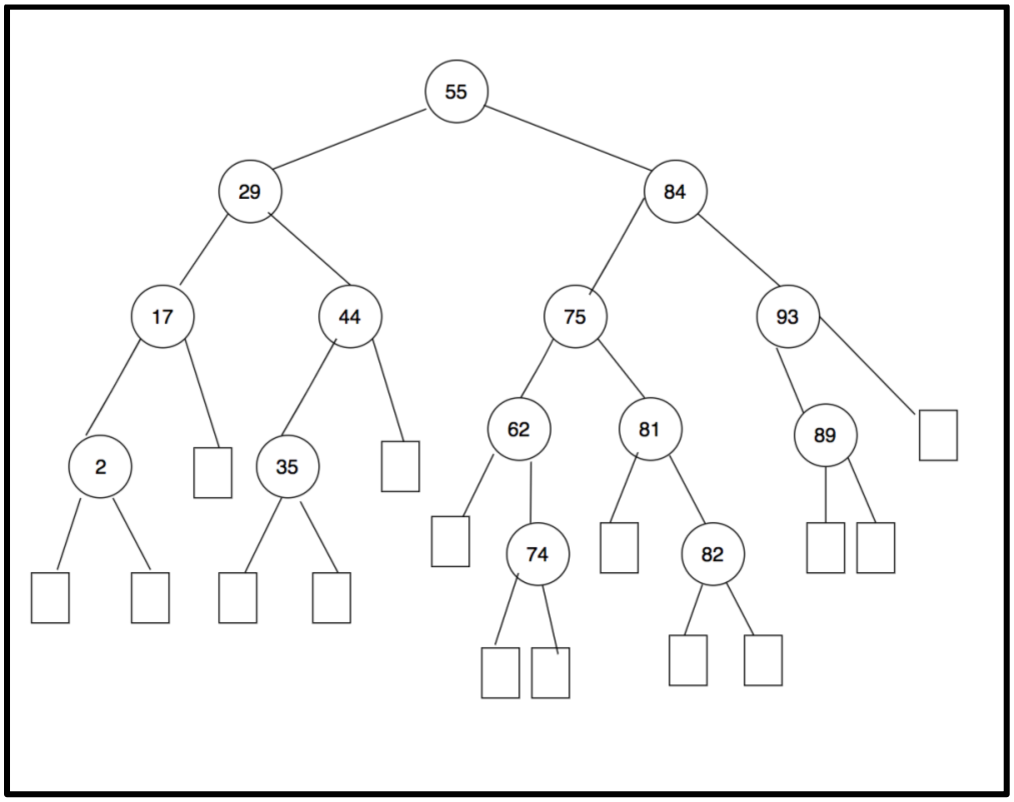


Now work with this properly balanced AVL tree.

ii) put(74)

74 will be placed in the right subtree of 62 because it is greater than 62. Now let's study the new height of the subtree rooted at 75. The left subtree has height 2 and the right subtree has height 2. Study the height of the subtree rooted at 84. The left subtree has height 3 and the right subtree has height 2. Now let's study the height of the tree rooted at 55. The left subtree has height 3 and the right subtree has height 4. So, we do not need to apply any rotations, because the height balance property of this AVL tree is not violated.

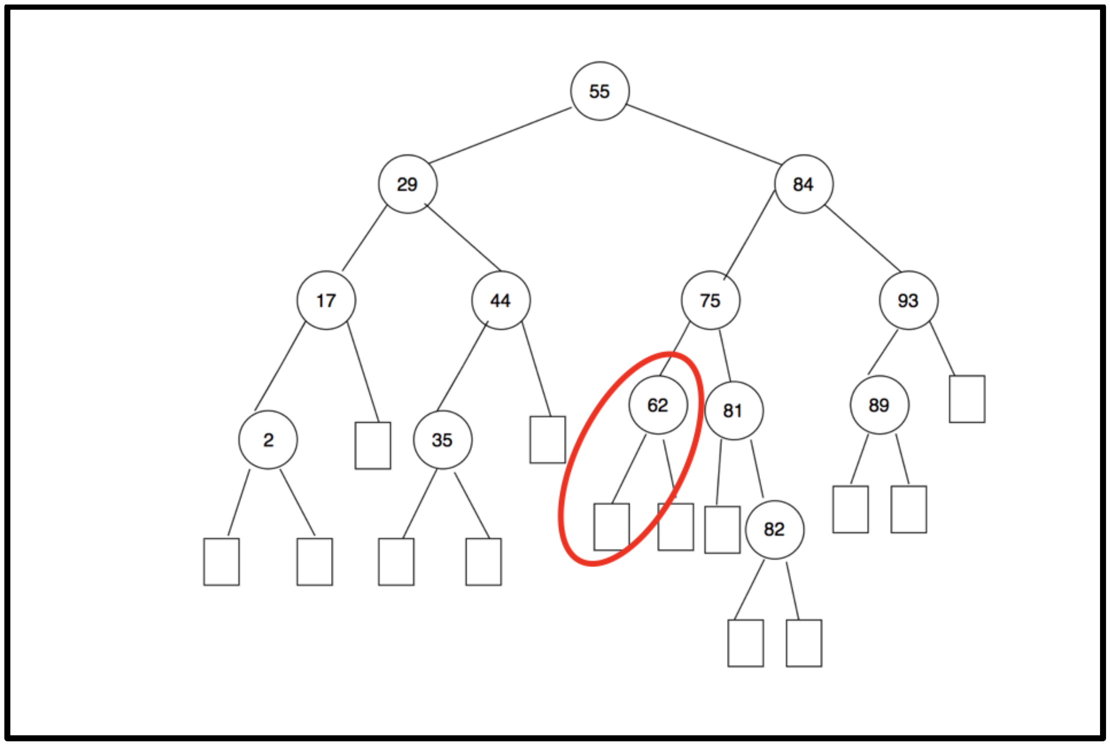
Time Complexity: Let n be the number of keys the AVL tree stores. Now, the height of an AVL tree that stores n keys is O(log n). The insertion depends on the height of the AVL tree, so put(74) runs in O(log n).

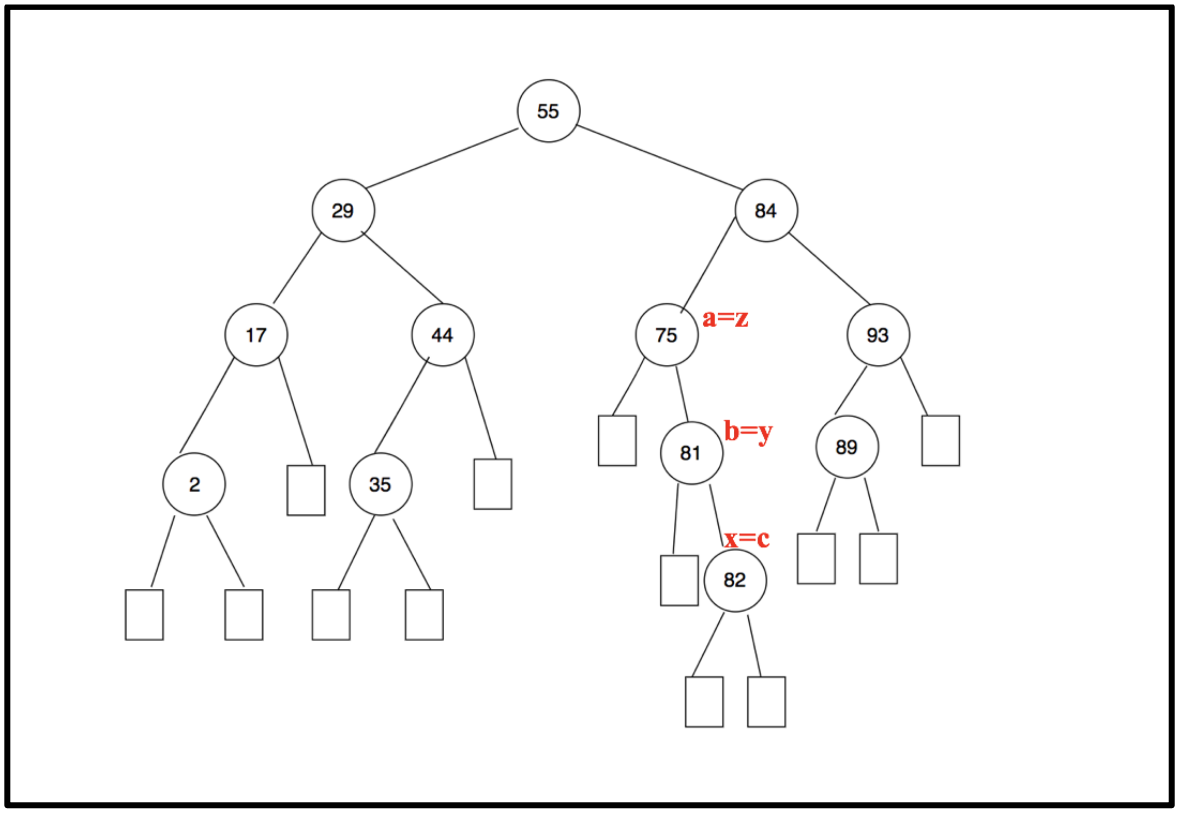


iii) remove(62)

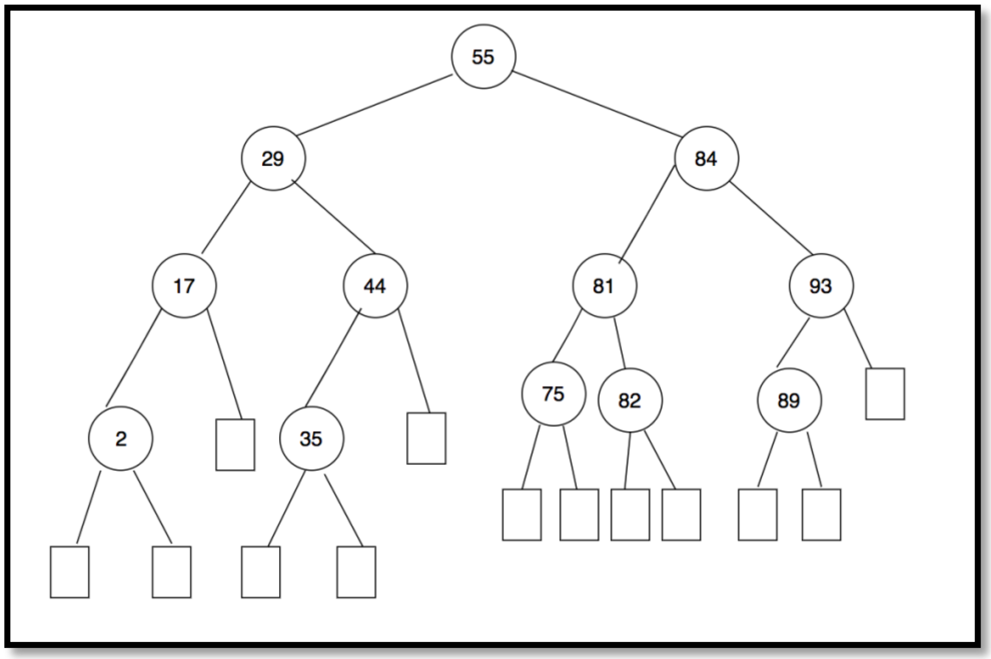
Since 62 has 2 leaf children, we delete 62 and a leaf child, and replace 62 with the leaf child.

Time Complexity: Let n be the number of keys the AVL tree stores. Now, the height of an AVL tree that stores n keys is O(log n). The deletion depends on the height of the AVL tree, so remove(62) runs in O(log n). Restructuring takes O(1) time.





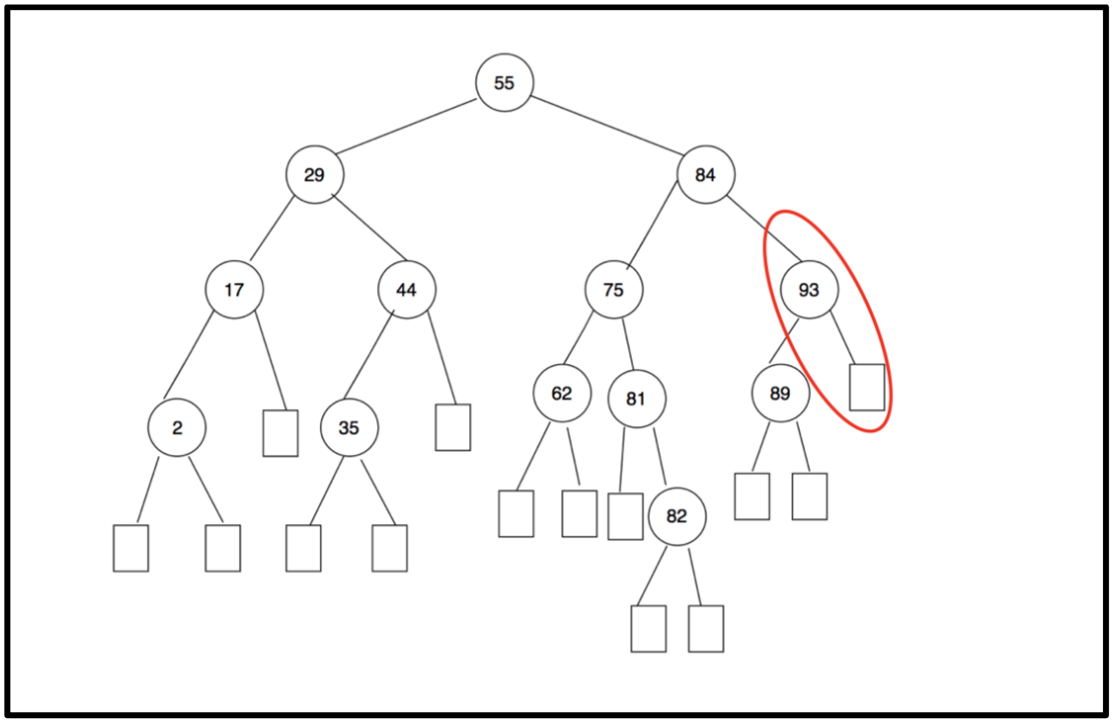
Let’s study the height of the subtree rooted at 75. The left subtree has height 0 and the right subtree has height 2. So we must restructure the tree rooted at 75. Z is the 1st encountered node causing unbalance (75). Y is the child of z with the higher height (81). X is the child of Y with the higher height (82). Perform a single rotation. In other words replace z with b, take a and c and make them children of b. Maintain in-order traversal of subtrees.

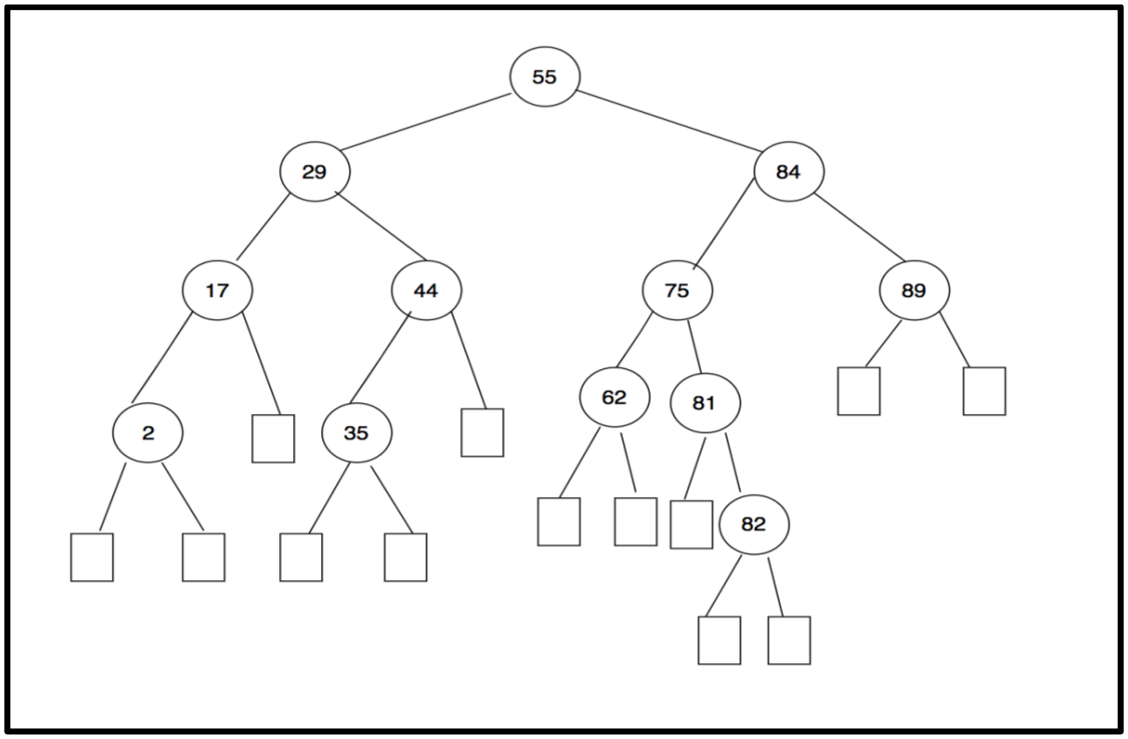


After we restructured the tree, let’s study the heights. The trees rooted at 75 and 82 have heights 1. The tree rooted at 81 has height 2 and the tree rooted at 93 has height 2. The tree rooted at 84 has height 3 and the tree rooted at 29 also has a height of 3. Clearly, the height-balance property is restored.

iv) remove(93)

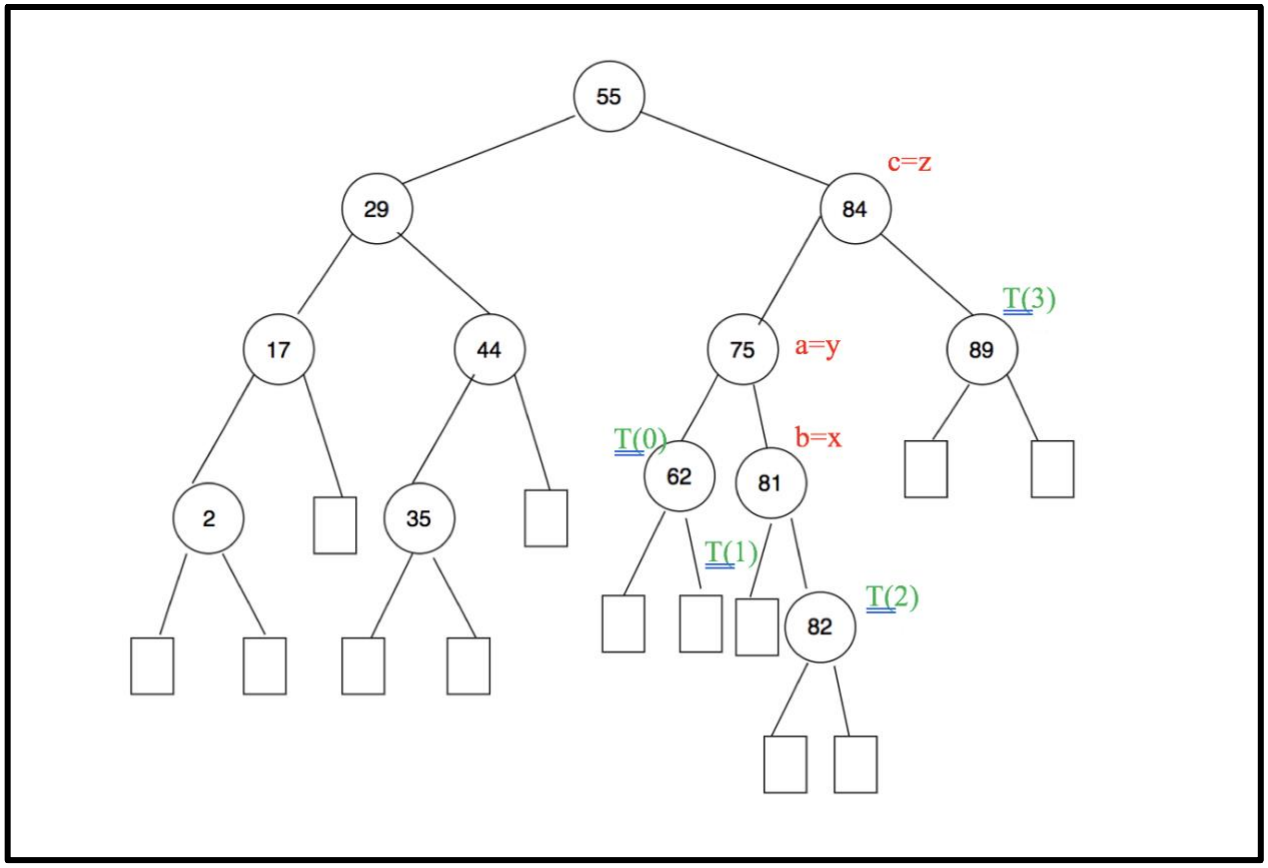
Since 93 has a leaf child, we delete 93 and the leaf child, and replace 93 with the internal node child (89).

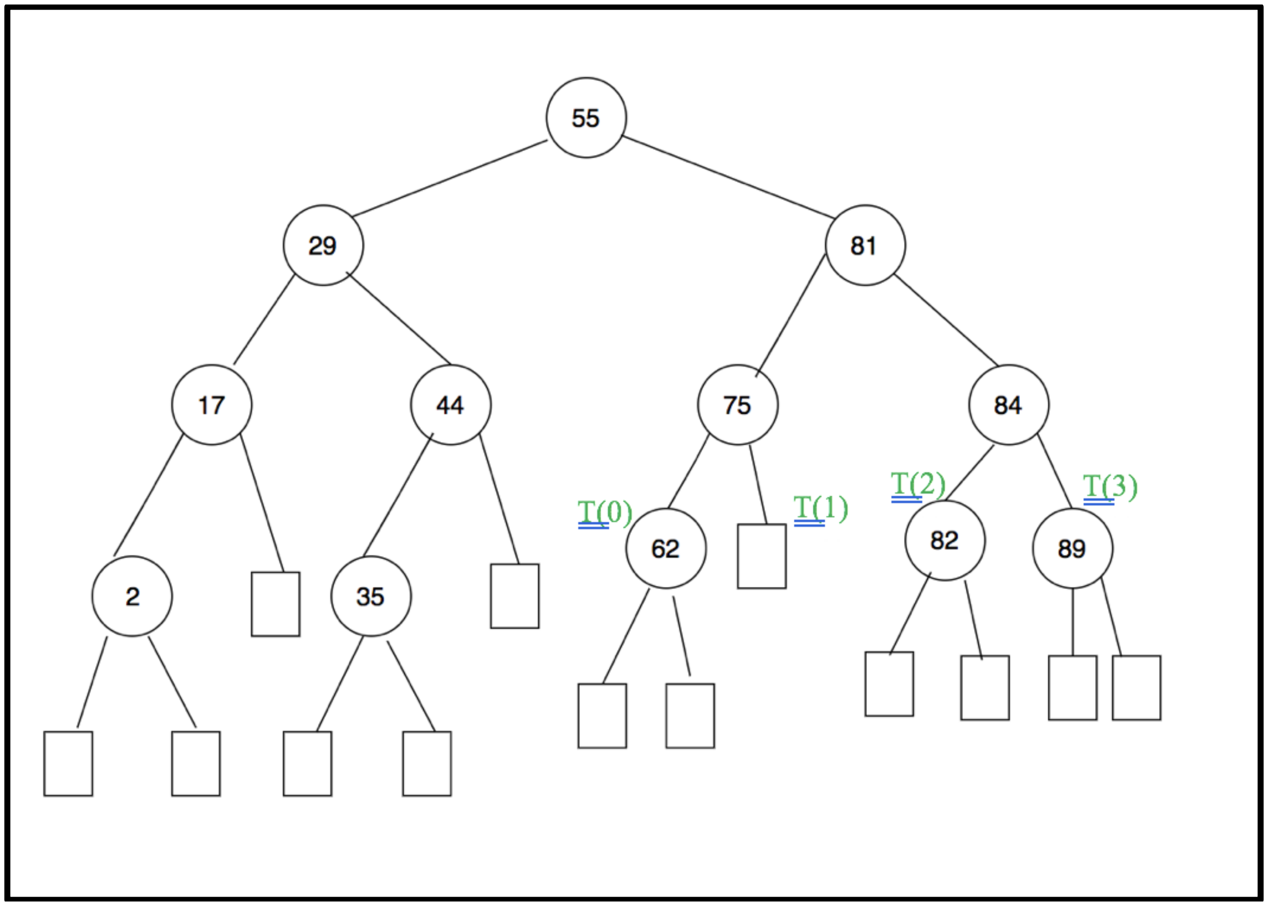




After deleting 93, let us study the heights of the tree. The tree rooted at 75 is not changed and has a height of 3. The tree rooted at 89, however, has a height of 1. There is a height difference of 2 between 89 and 75. Therefore, the tree rooted at 84 is no longer balanced.

Z is the 1st encountered node causing unbalance (84). Y is the child of z with the higher height (75). X is the child of Y with the higher height (81). Perform a double rotation. In other words replace z with b, take a and c and make them children of b. Maintain in-order traversal of subtrees.





Now, let’s study the height of this tree, after restructuring it.
The tree rooted at 62 has height 1, and its sibling has height 0.
The tree rooted at 82 and the tree rooted at 89 have heights 1.
The tree rooted at 75 has height 2 and the tree rooted at 84 has height 2.
The tree rooted at 81 has height 3, and the tree rooted at 29 is unchanged so has height 3.
The tree rooted at 55 has height 4.

The height balance property is restored.

Time Complexity: Let n be the number of keys the AVL tree stores. Now, the height of an AVL tree that stores n keys is O(log n). The deletion depends on the height of the AVL tree, so remove(93) runs in O(log n). Note that rotations run in O(1) so even if we performed one rotation it will not significantly affect the log n time complexity.

Programming Questions: Pseudocode

```
public class CVR

    declare integer variables threshold, keyLength, size, max_size
    size←0
    constant String alphaNumeric,
    constant String  arrays owners, brands
    records_1←new DoublyLinkedList()
    records_2←new TreeMap<String, Vehicle>()

    Algorithm CVR ()
        Input None
        Output None
        Default constructor

    Algorithm CVR ( int n)
        Input size of CVR
        Output None
        max_size←n

    Algorithm GreaterThanThreshold (size_pass)
        Input number of entries
        Output If we should use ADT TreeMap
        if   size_pass >= threshold then
            return true
        else
            return false



    Algorithm setThreshold (threshold_pass)
        Input threshold
    Output nothing
    if   threshold_pass < 100 or threshold_pass > 900000 then
            display error and exit program
    else
            threshold← threshold_pass

Algorithm getThreshold ()
    Input None
    Output threshold
    return threshold

Algorithm getKeyLength ()
    Input None
    Output keyLength
    return keyLength

Algorithm setKeyLength (length)
    Input key length
    Output nothing
    if   length < 10 or length > 17 then
            display error and exit program
    else
            keyLength← length


Algorithm generate (n)
    Input how many keys to generate
    Output nothing
    if   size+n>max_size then
            display error and return
    if  GreatherThanThreshold(n + size) then
```

```
        if   sequence_first=true then
            changeADT()
        for i←0 to n-1 do
            owner← owners[((int) (Math.random() * owners.length))]
            brand← brands[((int) (Math.random() * brands.length))]
            price←55555*Math.random()
            acc←new Stack<Date>()
            acc← new Date().randomDates()
            temp← new Vehicle (owner, brand, price, acc)
            temp_vin←generateVIN(keyLength)
            if   ! records_2.containsKey(temp_vin) then
                records_2.put(temp_vin, temp)
                increment size
            else if   records_2.containsKey(temp_vin) then
                decrement i
                continue
    else if  ! GreatherThanThreshold(n + size) then
        sequence_first←true
        for i←0 to n-1 do
            owner← owners[((int) (Math.random() * owners.length))]
            brand← brands[((int) (Math.random() * brands.length))]
            price←55555*Math.random()
            acc←new Stack<Date>()
            acc← new Date().randomDates()
            temp← new Vehicle (owner, brand, price, acc)
            temp_vin←generateVIN(keyLength)
            if   size=0 then
                records_1.addFirst(temp_vin, temp)
                increment size
            else if   ! records_1.contains(temp_vin) then
                records_1.addFirst(temp_vin, temp)
                increment size
            else if   records_1.contains(temp_vin) then
                decrement i
                continue
    sort()

Algorithm generateVIN (keyLength)
    Input length of key to be generated
    Output random key with inputted length
    sb←new StringBuilder()
    while   keyLength-- !=0 do
        ch← (int) (Math.random() * alphaNumeric.length())
        sb.append (alphaNumeric.charAt(ch))
    return sb.toString()

Algorithm allKeys ()
    Input nothing
    Output String LinkedList with keys lexicographically sorted
    if   GreatherThanThreshold(size) then
        keys←records_2.keySet()
        sorted.addAll(keys)

    else
        sorted←records_1.allKeys()
    return sorted

Algorithm sort ()
    Input nothing
    Output nothing, just sorts the CVR keys lexicographically
    if   ! GreatherThanThreshold(size) then
        records_1←records_1.sortList()
```

```
Algorithm add (key_pass, owner_pass, brand_pass, price_pass, accidents_pass)
    Input the VIN (key_pass) and the vehicle properties
    Output nothing
    if   size=max_size then
        Display error and return
    vehicle_pass←new Vehicle (owner_pass, brand_pass, price_pass, accidents_pass)
    if   GreatherThanThreshold(size) then
        if   records_2.containsKey(key_pass) then
            Display that key already is there
        else
            if sequence_first=true then
                changeADT()
            records_2.put(key_pass, vehicle_pass)
            increment size
    else if  ! GreatherThanThreshold(size) then
        sequence_first←true
        if records_1.contains(key_pass) then
            Display that key already is there
        records_1.addHere (key_pass, vehicle_pass)
        increment size


Algorithm remove (key_pass)
    Input the VIN whose vehicle will be deleted from CVR
    Output nothing
    if   GreatherThanThreshold(size) then
        if   records_2.remove(key_pass)!=null then
            decrement size
        else
```

```
                display error has occurred
        else if  ! GreatherThanThreshold(size) then
            if  records_1.delete(key_pass) then
                decrement size
            else
                display error has occurred

Algorithm getValue (key_pass)
    Input the VIN whose vehicle will be returned
    Output vehicle of the given VIN
    if  GreatherThanThreshold(size) then
        return records_2.get (key_pass)
    else
        return records_1.getValueFromKey (key_pass)

Algorithm nextKey (key_pass)
    Input the VIN
    Output the VIN after the given VIN
    if  GreatherThanThreshold(size) then
        if  records_2.higherEntry(key_pass) = null then
            return null
        return records_2.higherEntry(key_pass).getKey()
    else
        return records_1.nextKey(key_pass)
Algorithm prevKey (key_pass)
    Input the VIN
    Output the VIN before the given VIN
    if  GreatherThanThreshold(size) then
        if  records_2.lowerEntry(key_pass) = null then
            return null
        return records_2.lowerEntry(key_pass).getKey()
    else
        return records_1.prevKey(key_pass)

Algorithm changeADT ()
    Input nothing
    Output nothing
    while ! records_1.isEmpty() do
        key_temp←records_1.firstKey()
        vehicle_temp←records_1.firstValue()
        records_1.removeFirst()
        records_2.put(key_temp, vehicle_temp)
    sequence_first←false


Algorithm prevAccids (key_pass)
    Input the key whose previous accidents will be found
    Output previous accidents as an array list
    accidents←this.getValue(key_pass).getAccidents()
    tmpStack1 ←new Stack<Date>()
    tmpStack3←new Stack<Date>()


    while ! accidents.isEmpty() do
        tmp←accidents.pop()
        tmpStack3.push(temp)
        while (!tmpStack1.isEmpty() and (tmp.year > tmpStack1.peek().year
        or
        (tmp.year = tmpStack1.peek().year and tmp.month >
        tmpStack1.peek().month)
        or
        (tmp.year == tmpStack1.peek().year and tmp.month =
        tmpStack1.peek().month and tmp.day > tmpStack1.peek().day))) do
                accidents.push(tmpStack1.pop())
        tmpStack1.push(tmp)

temp←newStack<Date>()
while ! tmpStack1.isEmpty() do
        temp.push(tmpStack1.pop())

tmpStack2 ←new Stack<Date>()
while ! temp.isEmpty() do
        tmpStack2.push(temp.pop())
while ! tmpStack3.isEmpty() do
        accidents.push(tmpStack3.pop())
return tmpStack2
```

Programming Questions: CVR Report

How does the program work?

Before generating the keys, the user must set the size of the CVR, threshold, and the key length. Throughout the code the variable size keeps of track of how many entries are in the CVR. If the number of entries is less than the threshold, then we use a sequence: a double linked list. I chose to modify the doubly linked list class and make use of the node class for some of my methods. In fact, while reading the pseudocode, you may have noticed some unfamiliar methods that are being called with a doubly linked list. These methods were introduced to the DoublyLinkedList. Now, if the number of entries exceeds the threshold, then we must switch the ADT and use a TreeMap. This is already given by Java and I did not have to explicitly define the set and get methods for this class. Note that if we generate more keys than the threshold, then we would not have to use the sequence at all! However, if we start with a small number of entries, then exceed the threshold, the Boolean variable sequence_first will indicate that I have to switch my ADT using an auxiliary method called changeADT().

Why did you choose those Data Structures?

"if a CVR contains only a small number of entries (e.g., few hundreds), it might use less memory overhead but slower (sorting) algorithms. On the other hand, if the number of contained entries is large (greater than 1000 or even in the range of tens of thousands of elements or more), it might have a higher memory requirement but faster (sorting) algorithms"-assignment 3

In my program, the doubly linked list is a perfect match for the first data structure. Why? Well because the space needed is exactly equal to the number of entries stored in the CVR. In this case, it is at a bare minimum. And for the second data structure, a TreeMap is a perfect implementation because it provides an effective way of storing key-value pairs in a sorted order. In fact, I don't even have to sort the keys myself. The treemap automatically does that! For me, that was the main reason I favored the treemap over other data structures, like an AVL tree. In addition, the memory taken is quite low! We space needed is also exactly the same as the number of entries.

What is the time complexity of your methods? (*DoublyLinkedList)*

Let n be the number of entries in the CVR

operations applicable to a complete CVR <= O(n)
operations applicable to a single CVR entry <= O(n^2)

Method	**T.C**	**Justification**
setThreshold(Threshold)	O(1)	because we are simply initializing the variable threshold
setKeyLength(Length)	O(1)	because we are simply initializing the variable keylength
generate(n)	O(n^2)	Since we have a for loop the generate method runs in O(n) time. Note, however, that it would make sense to sort the elements after generating the keys and adding them. Since the sort() method runs in O(n^2) time, the generate method, overall, runs in O(n^2)
allKeys()	O(n)	To return all the keys will have to traverse over all the elements of the list and so this method runs in O(n)
add(key,value)	O(n)	We have to traverse over the elements of the list to know where to add the key while maintaining the lexicographic order. So the worst case scenario would be if the key were to be added at the very end (last in lexicographic order). In this case we'd have to traverse all the elements n , hence O(n)
remove(key)	O(n)	The worst case scenario here would be that we have to remove the last key in the list. In that case we would have to traverse over all the elements of the list (n), hence O(n)
getValues(key)	O(n)	The worst scenario would be that the key is the last element. Hence, we have to traverse over the entire list with a for loop O(n)
nextKey(key)	O(n)	The worst scenario would be that the key is the before-to-last element. Hence, we have to traverse over the entire array with a for loop O(n)
prevKey(key)	O(n)	The worst scenario would be that the key is the last element. Hence, we have to traverse over the entire array with a for loop O(n)
prevAccids(key)	O(n)	The worst case scenario would be that the key is the last element of the list. After we reach the last element we simply have to return the accidents of that key using a stack. Note that even if we use a while loop, returning the accidents of a key takes a constant time, because we have a constant number of accidents in every key, in my case, 3. Hence, time complexity is O(n). Of course, if we wanted to consider n' as the number of accidents, then the time complexity would be O(n'^2), (2 while loops) .

What is the time complexity of your methods? (*TreeMap)*

Method	**T.C**	**Justification**
setThreshold(Threshold)	O(1)	O(1) because we are simply initializing the variable threshold
setKeyLength(Length)	O(1)	O(1) because we are simply initializing the variable keylength
generate(n)	O(n log n)	Since we have a for loop the generate method runs n times. Time to insert first element = O(1) Time to insert second element = O(Log 1) = 0 = O(1) Time to insert third element = O(Log 2) Total time = Log 1 + Log 2 + Log 3 + ... + Log (n-1) And we don't have to sort the keys after that, that is automatic
allKeys()	O(1)	To return all the keys we call the method entrySet() which runs in O(1)
add(key,value)	O(1)	We call the method put (key,value) which runs in O(1)
remove(key)	O(1)	We call the method remove(key) which runs in O(1)
getValues(key)	O(1)	We call the method get(key) which runs in O(1)
nextKey(key)	O(1)	We call the method higherEntry(key).getKey(), which runs in O(1)
prevKey(key)	O(1)	We call the method lowerEntry(key).getKey(), which runs in O(1)
prevAccids(key)	O(1)	Note that even if we use a while loop, returning the accidents of a key takes a constant time, because we have a constant number of accidents in every key, in my case, 3. Hence, time complexity is O(1). Of course, if we wanted to consider n' as the number of accidents, then the time complexity would be O(n'^2), (2 while loops).

Comparison

Method	DoublyLinkedList	TreeMap	Meets Requirements?
setThreshold(Threshold)	O(1)	O(1)	Yes
setKeyLength(Length)	O(1)	O(1)	Yes
generate(n)	O(n^2)	O(n log n)	Yes
allKeys()	O(n)	O(1)	Yes
add(key,value)	O(n)	O(1)	Yes
remove(key)	O(n)	O(1)	Yes
getValues(key)	O(n)	O(1)	Yes
nextKey(key)	O(n)	O(1)	Yes
prevKey(key)	O(n)	O(1)	Yes
prevAccids(key)	O(n)	O(1)	Yes

Conclusion

Both the sequence and the ADT meet the time requirements of the assignments: *"operations applicable to a single CVR entry should be between O (1) and O (log n) but never worse than O(n). Operations applicable to a complete CVR should not exceed O(n^2)".*
Clearly, the treemap is a much faster implementation of the CVR. In fact most of its methods run in O(1) except for the generate method. The worst time complexity recorded is the generate method for the DoublyLinkedList, which runs in O(n^2).

Programming Questions: Array & Linked List

*Discuss how both the time and space complexity change for each of the methods above if the underlying structure of your CVR is an array or a linked list***?**

Case 1- Array

Let n be the number of entries of the CVR

Constructor
Time Complexity: O(1), because we are simply creating an initializing an array with the given length.
Space Complexity: O(n), because we are allocating space for all n elements of the array

Sort()

```
String array[] = new String[n];
Fill array with the elements of the CVR
for i ← 0 to n-1 do

    for j ← i + 1 to n-1 do

         if (array[i].compareTo(array[j]) > 0)then

              temp_string ← array[i]
              array[i] ← array[j]
              array[j] ← temp
```

Time Complexity: O(n^2), because 2 for loops are needed to sort the VINS lexiciographically
Space Complexity: O(n), because we are allocating space for all n elements of the array array

setThreshold(Threshold)
Time Complexity: O(1), because we are simply initializing the variable threshold
Space Complexity: O(1), because we are simply initializing one variable threshold

setKeyLength(Length)
Time Complexity: O(1), because we are simply initializing the variable keyLength
Space Complexity: O(1), because we are simply initializing one variable keyLength

generate(n)
Time Complexity: Like my code, I will require a for loop that runs from 0 to n-1, and generated a random VIN. Generating 1 random VIN takes O(1) time, and adding it to the array also takes O(1) time. However since we have a for loop the generate method runs in O(n) time.
Note, however, that it would make sense to sort the elements after generating the keys and adding them. Since the sort() method runs in O(n^2) time, the generate method, overall, runs in O(n^2).

Space Complexity: The generate method itself needs a constant space, because we use a limited number of variables. However, since we are calling the method sort(), technically speaking this method takes O(n) space.

allKeys()
Time Complexity: Assume that the array is already sorted lexicographically after generating the keys (see above). Now, to return all the keys will have to traverse over all the elements of the array and so this method runs in O(n).

Space Complexity: Assume that the array is already sorted lexicographically after generating the keys (see above). Now, to return all the keys will have to traverse over all the elements of the array and perhaps store in them in a sequence of some sort (array list, list, another array). Since this sequence will have the same space as the array itself, this method has O(n) space complexity.

add(key,value)
Time Complexity: The worst case scenario here would be that we have to add the key at the very beginning, and there are already the maximum number of elements in the array minus 1. In this case, we would have to shift all the elements of the array by 1 unit to the right, most easily with a for loop. Hence, O(n) time complexity.
Space Complexity: Since we can operate on the array itself, we do not need an auxiliary array. We simply need to allocate space for a constant number of variables, hence O(1).

remove(key)
Time Complexity: The worst case scenario here would be that we have to add the remove the key from the very beginning. In that case we would have to shift all elements of the array one unit to the left, most easily with a for loop. Hence O(n) time complexity. Note that even if the key is the last entry, we still have to traverse over the entire array: O(n).
Space Complexity: Since we can operate on the array itself, we do not need an auxiliary array. We simply need to allocate space for a constant number of variables, hence O(1).

getValues(key)
Time Complexity: The worst scenario would be that the key is the last element. Hence, we have to traverse over the entire array with a for loop O(n).
Space Complexity: Since we are using the array itself, we don't need an auxiliary data structure. We simply need to allocate space for a constant number of variables, hence O(1).

nextKey(key)
Time Complexity: The worst scenario would be that the key is the before-to-last element. Hence, we have to traverse over the entire array with a for loop O(n).
Space Complexity: Since we are using the array itself, we don't need an auxiliary data structure. We simply need to allocate space for a constant number of variables, hence O(1)

prevKey(key)

Time Complexity: The worst scenario would be that the key is the last element. Hence, we have to traverse over the entire array with a for loop O(n).
Space Complexity: Since we are using the array itself, we don’t need an auxiliary data structure. We simply need to allocate space for a constant number of variables, hence O(1)

prevAccids(key)
Recall: *Let n be the number of entries of the CVR, NOT the number of accidents of the key*

Time Complexity: The worst case scenario would be that the key is the last element of the array. After we reach the last element we simply have to return the accidents of that key using a stack. Note that even if we use a while loop, returning the accidents of a key takes a constant time, because we have a constant number of accidents in every key, in my case, 3. Hence, time complexity is O(n).
Space Complexity: We are allocating space in a stack for the accidents. However, we have a constant number of accidents in every key, in my case, 3. Hence the space complexity is constant regardless of the inputted key, so O(1).
Of course, if we wanted to consider n’ as the number of accidents, then the time complexity would be O(n’^2), and the space complexity would be O(n’).

<u>Case 2- LinkedList</u>

Let n be the number of entries of the CVR

Constructor
Time Complexity: O(1), because we are simply creating an initializing a linked list with the given length.
Space Complexity: O(n), because we are allocating space for all n elements of the list

Sort()
Time Complexity: O(n^2), because 2 for loops are needed to sort the VINS lexiciographically
Space Complexity: O(1), because we are working on the list itself.
(see code below in my java files: class DoublyLinkedList)

```
if (header == null) {
        return this;
} else {
        for (current = header.next; current.next != null && current.key != null; current = current.next) {
        for (index = current.next; index != null && index.key != null; index = index.next) {

                if (current.getKey().compareTo(index.getKey()) > 0) {
                        key_tempo = current.getKey();
                        value_tempo = current.getValue();

                        current.key = index.getKey();
                        current.value = index.getValue();

                        index.key = key_tempo;
                        index.value = value_tempo;


                }


        }
}
        return this;
```

setThreshold(Threshold)
Time Complexity: O(1), because we are simply initializing the variable threshold
Space Complexity: O(1), because we are simply initializing <u>one</u> variable threshold

setKeyLength(Length)

Time Complexity: O(1), because we are simply initializing the variable keyLength
Space Complexity: O(1), because we are simply initializing <u>one</u> variable keyLength

generate(n)
Time Complexity: Like my code, I will require a for loop that runs from 0 to n-1, and generated a random VIN. Generating 1 random VIN takes O(1) time, and adding it to the list also takes O(1) time. However since we have a for loop the generate method runs in O(n) time.
Note, however, that it would make sense to sort the elements after generating the keys and adding them. Since the sort() method runs in O(n^2) time, the generate method, overall, runs in O(n^2).
Space Complexity: The generate method itself needs a constant space, because we use a limited number of variables. So O(1) space. We are allocating space in a stack for the accidents. However, we have a constant number of accidents in every key, in my case, 3. Hence the space complexity is constant regardless of the inputted key, so O(1).

allKeys()
Time Complexity: Assume that the list is already sorted lexicographically after generating the keys (see above). Now, to return all the keys will have to traverse over all the elements of the list and so this method runs in O(n).

Space Complexity: Assume that the list is already sorted lexicographically after generating the keys (see above). Now, to return all the keys will have to traverse over all the elements of the list and perhaps store in them in a sequence of some sort (array list, list, array). Since this sequence will have the same space as the array itself, this method has O(n) space complexity.

add(key,value)
Time Complexity: We have to traverse over the elements of the list to know where to add the key while maintaining the lexicographic order. So the worst case scenario would be if the key were to be added at the very end (last in lexicographic order). In this case we'd have to traverse all the elements n , hence O(n)
Space Complexity: Since we can operate on the list itself, we do not need an auxiliary list. We simply need to allocate space for a constant number of variables, hence O(1).

remove(key)
Time Complexity: The worst case scenario here would be that we have to remove the last key in the list. In that case we would have to traverse over all the elements of the list (n), hence O(n) time complexity. Unlike the array, we do not have to shift any elements here, just "redirect" the header and trailer, if necessary.
Space Complexity: Since we can operate on the list itself, we do not need an auxiliary list. We simply need to allocate space for a constant number of variables, hence O(1).

getValues(key)
Time Complexity: The worst scenario would be that the key is the last element. Hence, we have to traverse over the entire list with a for loop O(n).

Space Complexity: Since we are using the list itself, we don't need an auxiliary data structure. We simply need to allocate space for a constant number of variables, hence O(1).

nextKey(key)
Time Complexity: The worst scenario would be that the key is the before-to-last element. Hence, we have to traverse over the entire list with a for loop O(n).
Space Complexity: Since we are using the list itself, we don't need an auxiliary data structure. We simply need to allocate space for a constant number of variables, hence O(1)

prevKey(key)
Time Complexity: The worst scenario would be that the key is the last element. Hence, we have to traverse over the entire list with a for loop O(n).
Space Complexity: Since we are using the list itself, we don't need an auxiliary data structure. We simply need to allocate space for a constant number of variables, hence O(1)

prevAccids(key)
Recall: *Let n be the number of entries of the CVR, NOT the number of accidents of the key*

Time Complexity: The worst case scenario would be that the key is the last element of the list. After we reach the last element we simply have to return the accidents of that key using a stack. Note that even if we use a while loop, returning the accidents of a key takes a constant time, because we have a constant number of accidents in every key, in my case, 3. Hence, time complexity is O(n).
Space Complexity: We are allocating space in a stack for the accidents. However, we have a constant number of accidents in every key, in my case, 3. Hence the space complexity is constant regardless of the inputted key, so O(1).
Of course, if we wanted to consider n' as the number of accidents, then the time complexity would be O(n'^2), and the space complexity would be O(n').