Homework 3

Name: Amit Makashir

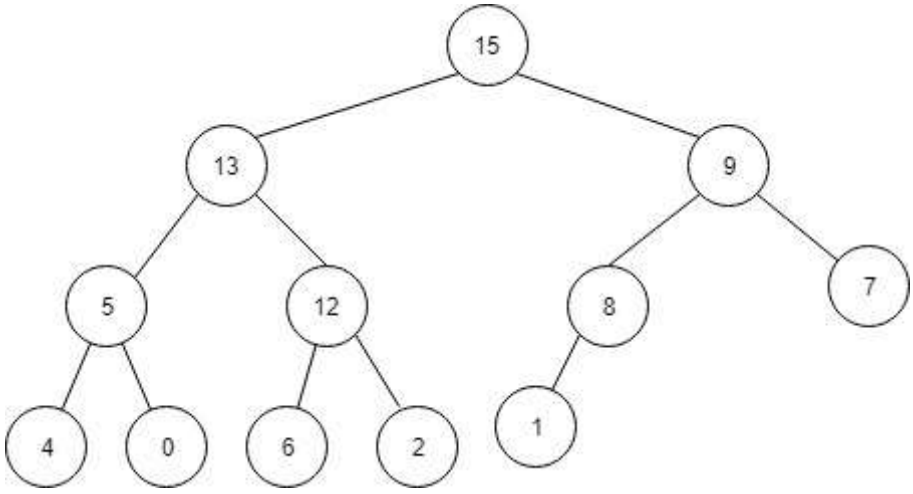
IU username: abmakash

Q1.

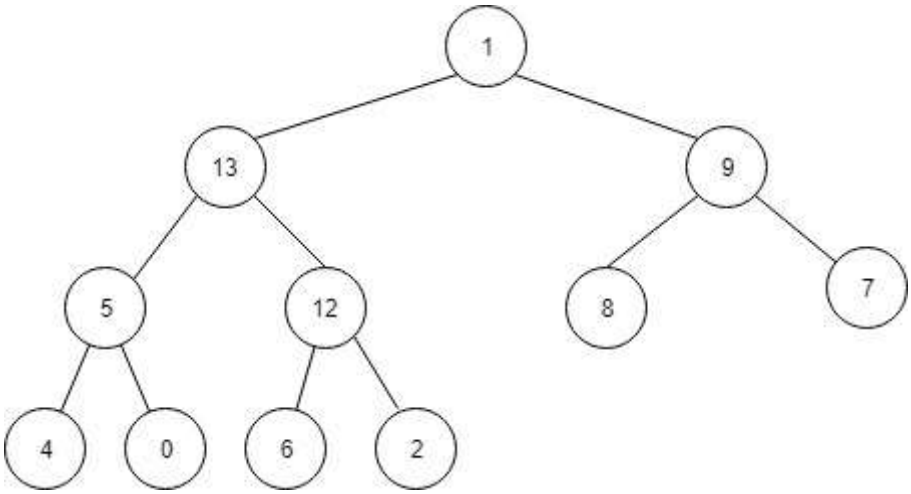
Illustrate the operation of extracting the maximum element on the binary heap A=<15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>;

Solution:

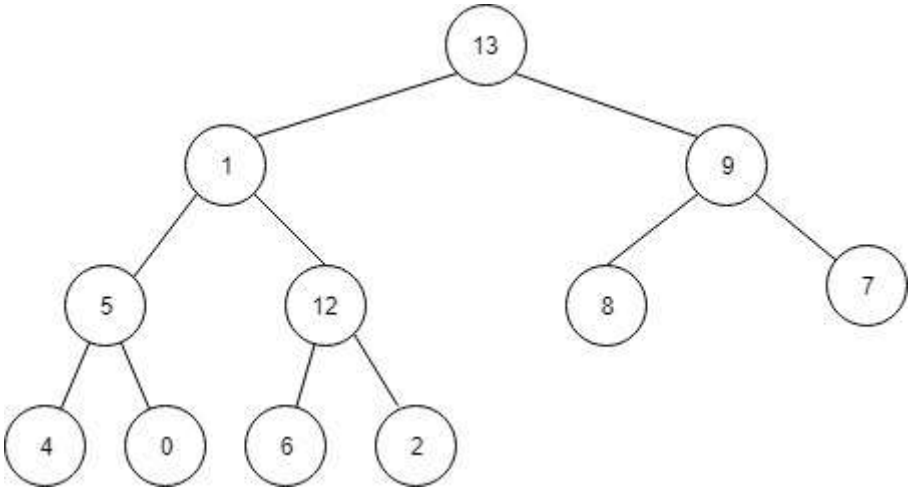
1. The root node in a max binary heap is the maximum element in the heap



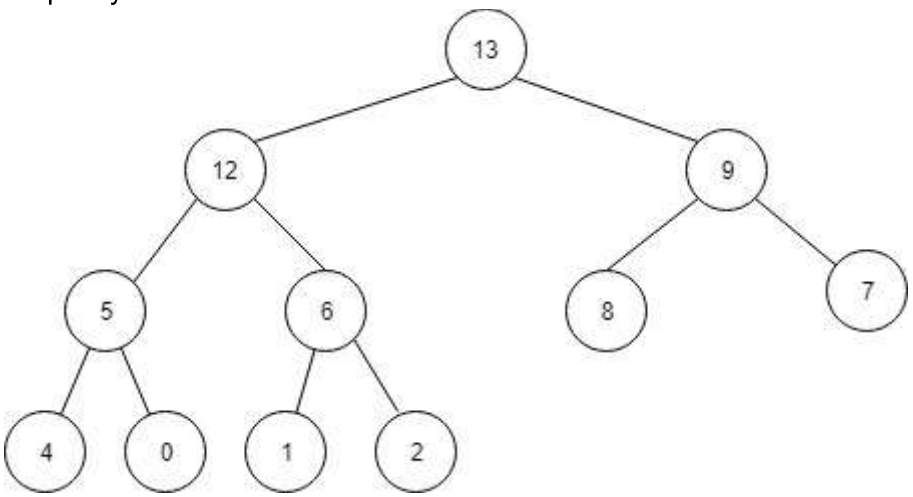
2. Extract the root node and swap it with the last element in the heap, in this case 1. Now delete this element



3. We swapped the position, but now the heap condition is not satisfied at some points. 1 less than 13 and 9 and yet is the parent of these elements. Let's do sift down operation to restore this heap. We compare 1 with 13 and 9, as 13 is largest amongst these, we swap 1 with 13



4. Now amongst 5 and 12, 12 is greater so we swap 1 with 12. Then amongst 6 and 2, 6 is greater so swap 1 by 6.



Q2

You are given an array of n elements, and you notice that some of these elements are duplicates; that is they appear more than one time in the array. Devise an algorithm to remove all duplicates from the array in time O(n log n).

Solution:

We can build a min-heap out of the array. We can extract the min element and push it to a new array, but only push if the last element pushed into this array was not the current element.

```
def RemoveDuplicates(array):
    # Create a min-heap of the array
    minheap = Heap.buildHeap(array)

    # Create an empty array where non-duplicates would be stored
    unique_list = []

    for i -> [0,length(array)-1]:
        # Extracts the root node, deletes it and does the sift down
        min_element = minheap.extractMin()

        # For the first iteration just push it to the unique list
        if i == 0:
            unique_list.append(min_element)

        # If the last element pushed in the unique_list array
        # is not this element, push this element
        else:
            if unique_list[-1] != min_element:
                unique_list.append(min_element)

    return unique_list
```

Q3

A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

Solution:

Let's analyse the amortized cost per operation in two parts:

1. If i^{th} the operation is not an exact power of 2, it requires 1 cost. Therefore, time complexity for such operations would be
$$\sum_{i=1}^{N} 1 = N$$
2. For the other operations, the time complexity would be:
$$= \sum_{i=1}^{log_2 N} 2^i$$

(solving for the sum of a geometric progression with r (common ratio) = 2, a_1 =2)

$$= 2(1 - 2^{(log_2 N)})/(1 - 2)$$
$$= 2(2^{(log_2 N)} - 1)$$
$$= 2(N - 1)$$

If we add these two costs, we would get:

$$(N + 2(N-1)) = 3N - 2 = O(n)$$

To calculate the amortized cost per operation, we will divide by N to get $O(1)$

Q4

A sequence of stack operations is performed on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made for backup purposes. Show that the cost of n stack operations, including copying the stack, is O(n) by assigning suitable amortized costs to the various stack operations.

Solution:

Let's say that it takes at the most "k" cost for copying the stack after k operations. (As the length of the stack never exceeds k)

1. The stack operations (Push, Pop, Multipop) take constant time per operation, Therefore time complexity for N such operations would be:
$$\sum_{i=1}^{N} O(1) = O(n)$$
2. After k operations we will copy this stack, so the time complexity for this operation would be:
$$\sum_{i=1}^{N/k} O(k) = (N/k) * O(k) = O(n)$$

Thus, the total cost of n stack operations, including copying the stack, is $O(n)$

Another approach would be to assign a credit of 2 units to each operation in the stack while the actual cost of that operation would 1 unit.

1. If we do k stack operations on the stack, the total credits would $2k - k = k$.
2. After this point, if we have to copy the entire stack, it would incur a cost of k. And our total credits would be 0.
3. Thus, we can say that, the stack operations require $O(1)$ time per operation. For N operations it would require $O(n)$

Q5

Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Show how to implement a counter as an array of bits so that any sequence of increment or reset operations takes time O(n) on an initially zero counter

Solution:

Pseudocode for the incrementing and resetting operations is given below:

```
def increment(array,a):
    i = 0

    # Make all the bits "0" till you find a "0"
    while i < len(array) and array[i] == 1:
        array[i] = 0
        i += 1

    if i < len(array):
        array[i] = 1

    a += 1
    return array,a


def reset(array,a,b):
    # Iterate through all the elements and make them "0"
    maxbits = log2(a-b+1) + 1
    for i in range(int(maxbits)):
        array[i] = 0

    b += 1
    return array,b

bits = [0,0,0,0,0,0,0,0]
a = 0 # No of increments
b = 0 # No of resets

bits,a = increment(bits,a)  # bits => [0,0,0,0,0,0,0,1], a=1
bits,a = increment(bits,a)  # bits => [0,0,0,0,0,0,1,0], a=2
bits,a = increment(bits,a)  # bits => [0,0,0,0,0,0,1,1], a=3
bits,a = increment(bits,a)  # bits => [0,0,0,0,0,1,0,0], a=4
bits,b = reset(bits,a,b)    # bits => [0,0,0,0,0,0,0,0], b=1
```

Time analysis:

a = # of increment operations (takes constant cost per operation)

b = # of resets (takes **k** cost, where k is the number of bits it has to traverse)

The total number of bits involved can be written as $k = log_2(a - b + 1) + 1$.

We know, $a + b = n$

The total cost would be:

$$
\begin{aligned}
&= a * 1 + b * k \\
&= n - b + b * (log_2(a - b + 1) + 1) \\
&= n - b + b * (log_2(n - b - b + 1) + 1)
\end{aligned}
$$

$$= n - b + b * (log_2(n - 2b + 1) + 1)$$

The upper bound of this function depends on b and the upper bound for b is n/2

$$= n/2 + n/2 * log_2(n - n + 1) + n/2$$
$$= n + n/2 * log_2(1)$$
$$= n$$

Therefore, any sequence of increment or reset operations takes time $O(n)$ on an initially zero counter.

Q6

Assuming that we begin with a stack with s_0 objects and after executing n of the stack operations PUSH, POP, and MULTIPOP, the stack ends up with s_n objects, we claim that the total cost of such operations is no more than $2n$ + s_0 - s_n . Please explain why.

Solution:

We can perform 3 different types of operations:

1. Multipop - let's say we do **a** number of Multipop operations (Time complexity: $O(k)$)
2. Pop - we do **b** pop operations (Time complexity: $O(1)$)
3. Push - we do **c** push operations (Time complexity: $O(1)$)

So, our total costs would be:

$$= a * k + b * 1 + c * 1$$
$$..eq(1)$$

We know the total operations are $a + b + c = n$

We also that difference of number of pushed elements and popped elements would be $(s_n - s_0)$

$$c - (a * k + b) = s_n - s_0$$
$$a * k + b = c - s_n + s_0$$
$$..eq(2)$$

Substituting eq(2) in eq(1), we get:

$$= 2c + s_0 - s_n$$

Thus, the total cost depends on **c** i.e the number of push operations. The upper bound on c is the total number of operations i.e **n** (there are only push operations, no pop or multipop).

Therefore, the total cost can never exceed $2n + s_0 - s_n$

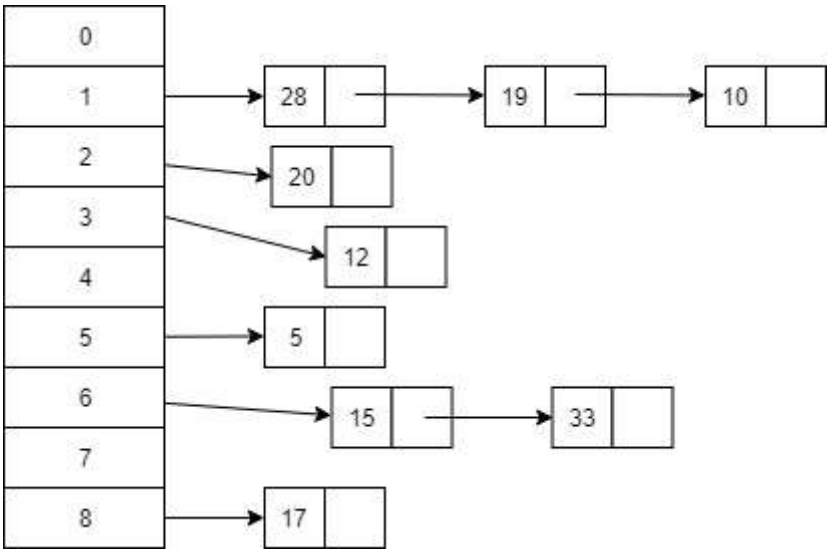
Q7

Illustrate the insertion of the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table has 9 slots, and let the hash function be h(k) = k mod 9.

Solution:

1. 5 mod 9 = 5. Let's place 5 in the 5th slot
2. 28 mod 9 = 1. Let's place 28 in the first slot
3. 19 mod 9 = 1. Let's place 19 in the first slot after 28
4. 15 mod 9 = 6. Let's place 15 in the 6th slot
5. 20 mod 9 = 2. Let's place 2 in 2nd slot
6. 33 mod 9 = 6. Let's place 33 in 6th slot
7. 12 mod 9 = 3. Let's place 12 in 3rd slot
8. 17 mod 9 = 8. Let's place 17 in 8th slot
9. 10 mod 9 = 1. Let's place 10 in first slot

Below is an image representing what the Hashtable might look like after inserting these keys

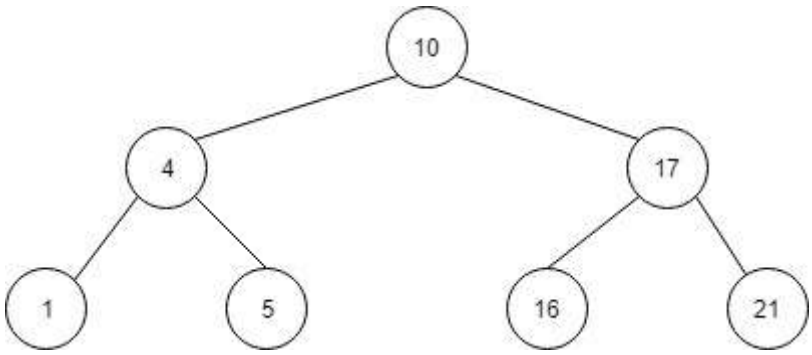


Q8

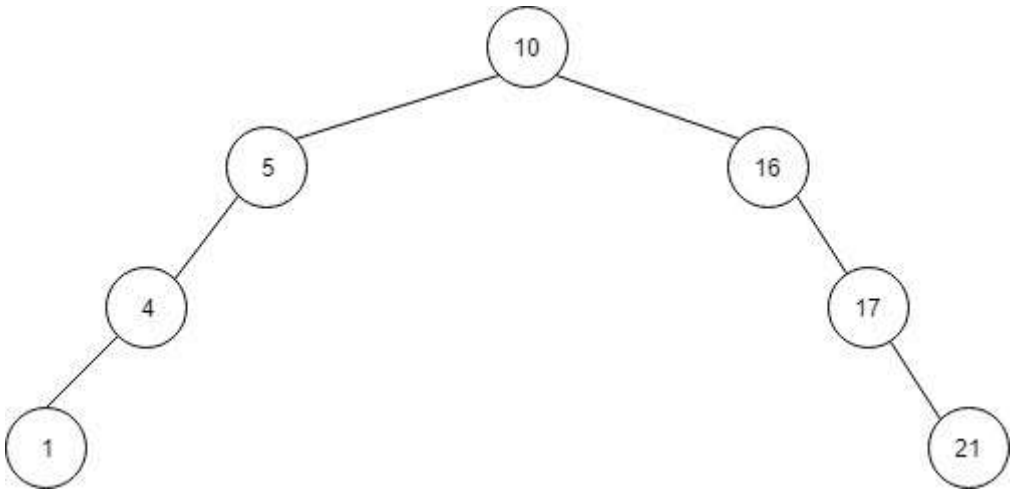
For the set of {1, 4, 5, 10, 16, 17, 21} of keys, draw binary search trees of height 2, 3, 4, 5, and 6.

Solution:

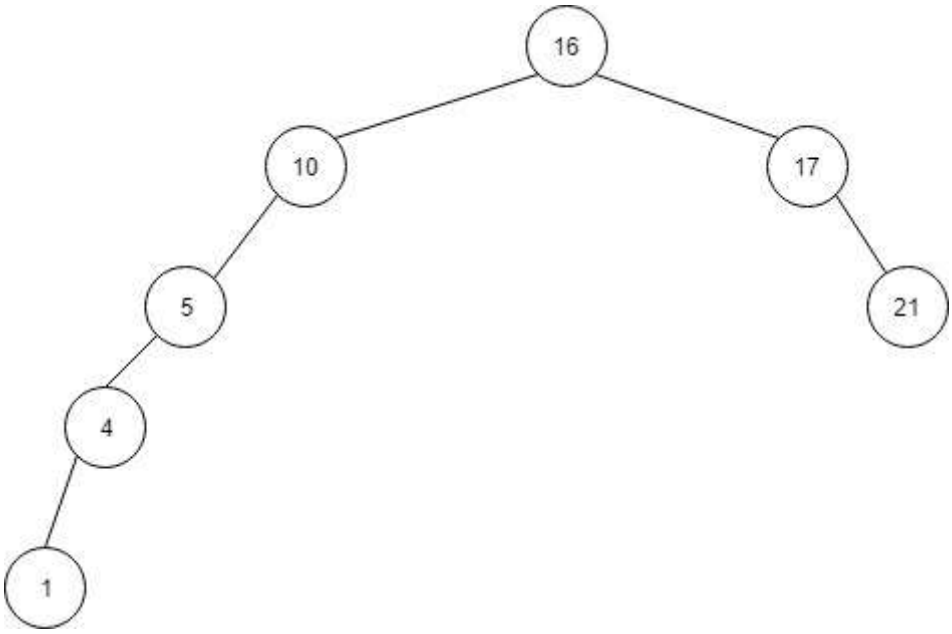
Depth = 2:



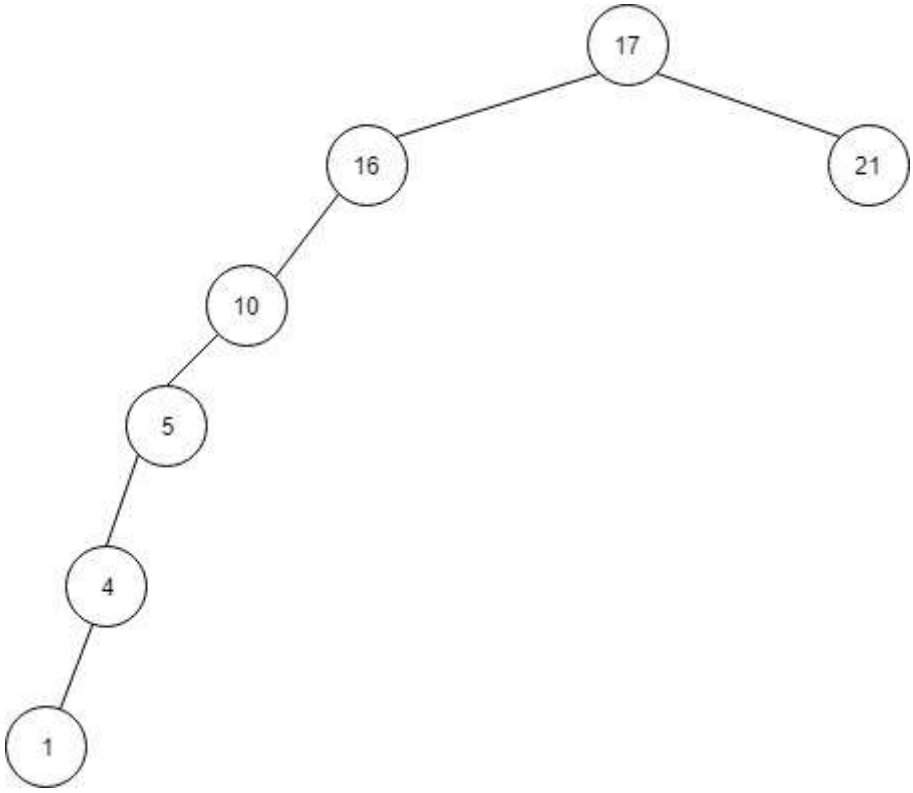
Depth = 3:



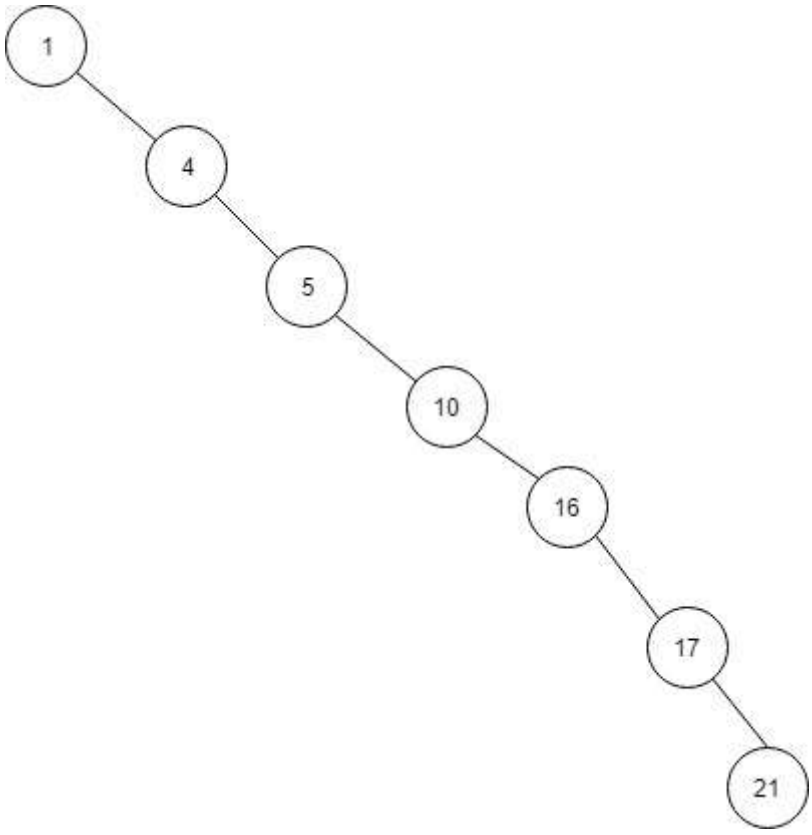
Depth = 4:



Depth = 5:



Depth = 6:



Q9

Suppose that the search for the key k in a binary search tree ends up in a leaf. Consider three sets: 1) the keys to the left of the search path; 2) the keys on the search path; 3) the keys to the right of the search path. One claims that any three keys a, b and c from these three sets, respectively (say a from 1, b from 2 and c from 3), it is always true that a ≤ b ≤ c. Is this claim true? Explain your answer.

Solution:

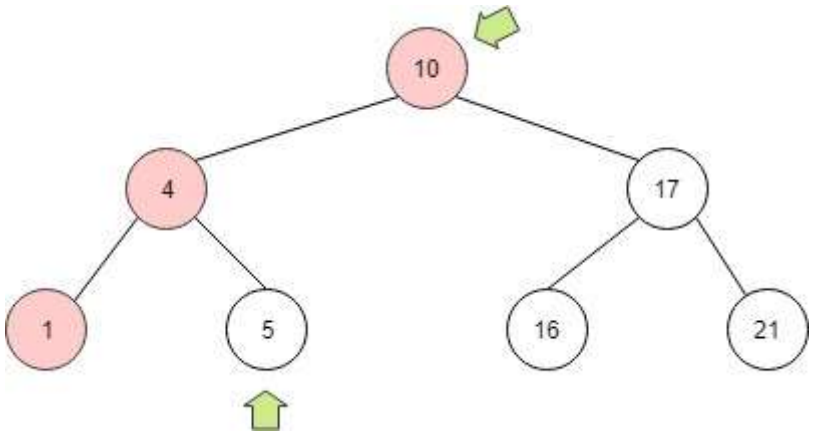
In the below image, let's say we are searching for **"1"**.

The red nodes are the search path for us in this case i.e they belong to set 2.

All the other nodes in this case belong to set 3.

5 belongs in set 3 and 10 belongs to set 2, but 5 is not greater than or equal to 10.

Therefore, it is not always true that $a <= b <= c$



Q10

We can sort a given set of n numbers by first building a binary search tree containing these numbers (using Tree_insert algorithm repeatedly to insert the numbers one by one) and then printing the number in an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

Solution:

There are two parts to this sorting algorithm:

1. Building the Binary search tree by inserting elements one after the other
2. Traversing (and printing) these numbers in an inorder tree walk

Part 2 requires us to visit every node and print in a particular sequence. This process take $O(n)$ time complexity. Part 1 inserts an element in the BST which takes $O(logn)$ (if the tree is balanced). We do this n times, so the complexity becomes $O(nlogn)$. This is the best case running time for this algorithm.

If the array is sorted (ascending or descending) the generated binary search tree would be heavily imbalanced and would look like a linked list. The insertion operation in such a tree takes $O(n)$. Inserting n elements thus would require $O(n^2)$.

For the best case, the array would have the following sequence:

1. The first element of the array is the median of the entire array
2. Imagine the entire array being sorted. The left sub-array would be elements smaller than median. The second element should be median of this sub-array.
3. Similarly, third element would median of the right sub-array.
4. All the other elements occur in the similar pattern.

This will create a balanced BST and would make the time complexity of the algorithm $O(nlogn)$