1.

(1)

O(lg(n)).

In a binary search tree, all the nodes are well sorted in a specific order (left child< parent < right child), therefore the length(number of nodes passed through) of our search path can only be equal to or smaller than that from the root to the furthest leaf node (since in this order, we will never need to go back while searching for a specific number).

In this case, the tree is perfectly balanced and full, so the length of the path from the root to the furthest leaf node is the height of the tree h, which is equal to lg(n+1) where n represents the total number of the nodes in the tree (according to the equation of the of a geometric series). Therefore, the big-O search time is O(lg(n)).

(2)

O(n).

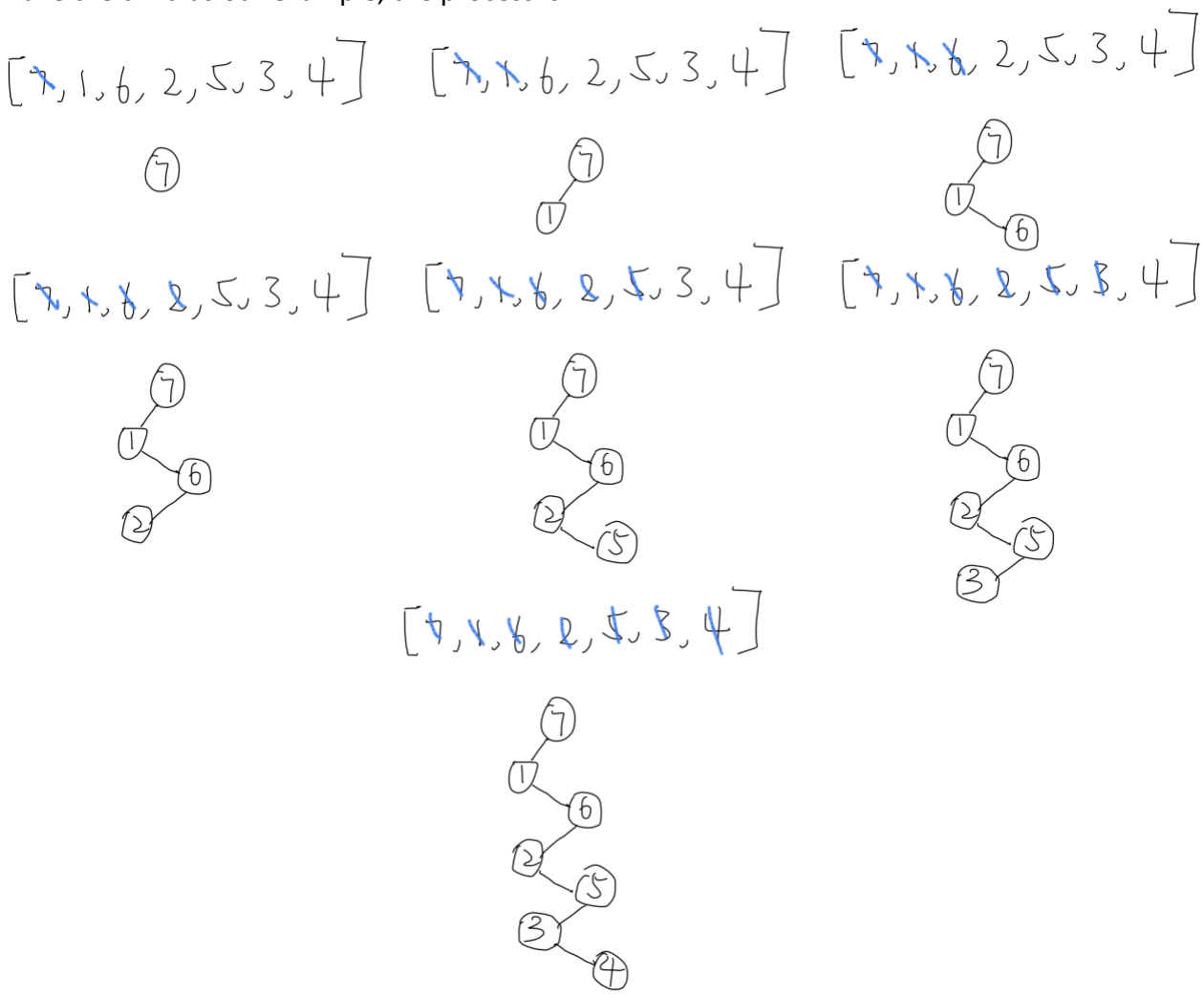
Similarly, if it is not balanced and full, the worst case is when every non-leaf node only has one child. Then the length of the path from the root to the furthest leaf node is equal to n. As explained in (1), then the big-O search time is O(n).

2.

The data set in which the numbers are monotone increasing/decreasing or in other special patterns could create an unbalanced binary search tree. For example:

[1, 2, 3, 4, 5, 6, 7]
[7, 6, 5, 4, 3, 2, 1]
[7, 1, 6, 2, 5, 3, 4]

Take the third as our example, the process is:



3.

Rules:

Rule (0) if the item I am deleting has no child, or is the root with one child:

Directly delete it.

Rule (1) else if the item I am deleting is not the root and has 2 children:

Do the same as rule (2).

Rule (2) else if the item I am deleting is not the root and only has a right child:

Insert the smallest item of the right subtree into the deleted position. Then regard this process as deleting the inserted item from its original position. Recursively do one of (0)-(3) according to this item's property.

Rule (3) else if the item I am deleting is not the root and only has a left child:

Insert the biggest item of the left subtree into the deleted position. Then regard this process as deleting the inserted item from its original position. Recursively do one of (0)-(3) according to this item's property.

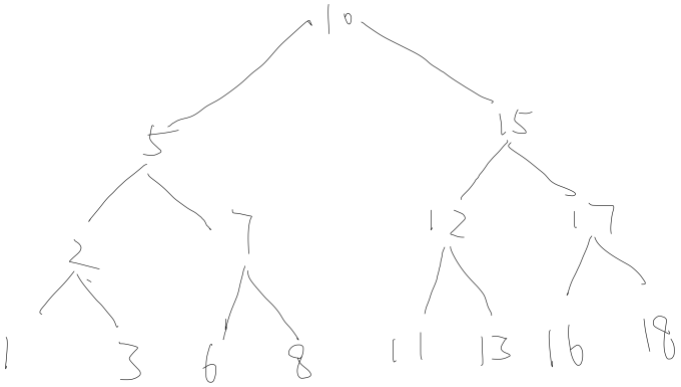
Rule (4) else if the item I am deleting is the root with 2 children:

Do the same as rule (1).

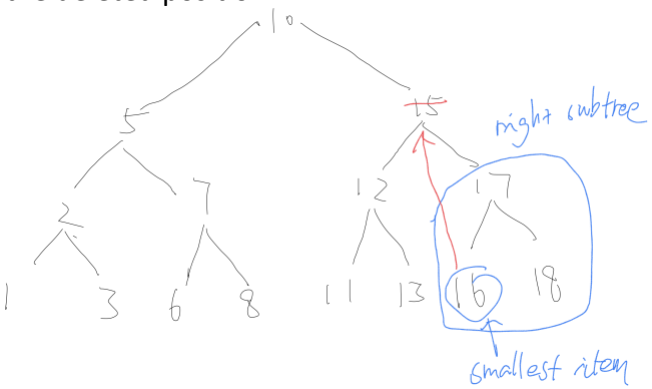
Instances are on the following pages.

Instances:

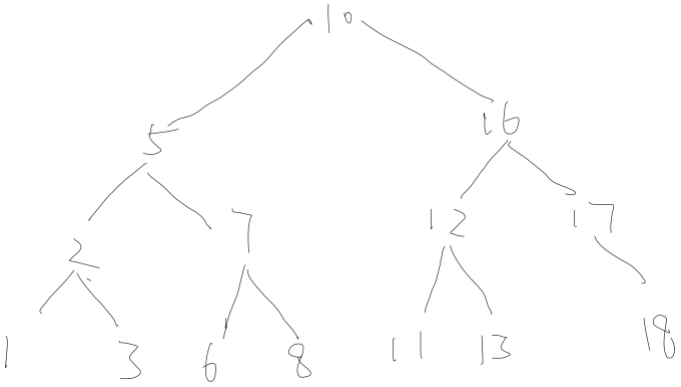
Instance 1, the item you're deleting has 2 children -- delete 15 from the tree below.



According to rule (1)->(2), remove 15 and Insert the smallest item(16) of the right subtree into the deleted position.



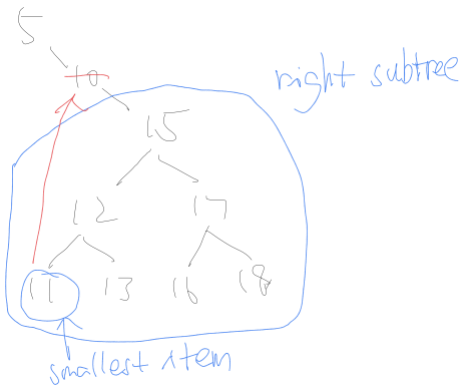
=>



Since item 16 in its original position conforms to rule (0), nothing else needs to be done. So, the whole process is completed.

Instance 2, the item you're deleting has a right child -- delete 10 from the tree below.

According to rule (2), remove 10 and Insert the smallest item(11) of the right subtree into the deleted position.

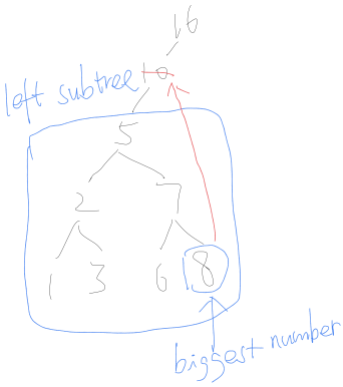


=>

Since item 11 in its original position conforms to rule (0), nothing else needs to be done. So, the whole process is completed.

Instance 3, the item you're deleting has a left child -- delete 10 from the tree below.

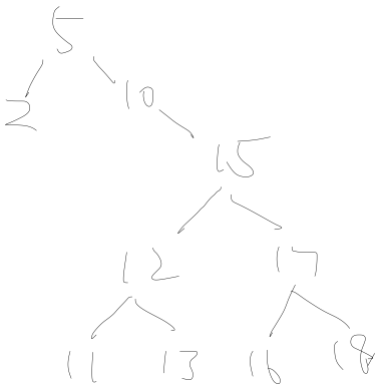
According to rule (3), remove 10 and Insert the biggest item(8) of the left subtree into the deleted position.



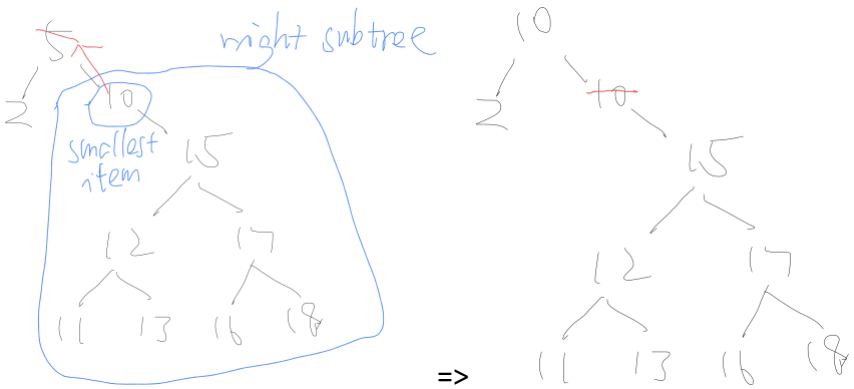
=>

Since item 8 in its original position conforms to rule (0), nothing else needs to be done. So, the whole process is completed.

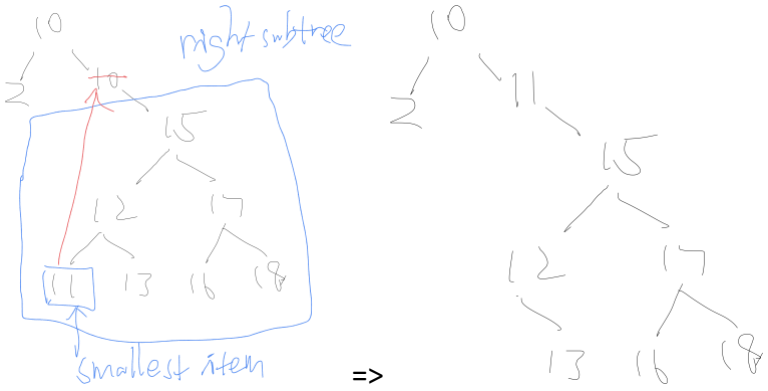
Instance 4, the item you're deleting is the root with 2 children -- delete 5 from the tree below.



According to rule (4)->(1)->(2), remove 5 and Insert the smallest item(10) of the right subtree into the deleted position.



Since item 10 in its original position conforms to rule (2), remove 10 and Insert the smallest item(11) of the right subtree into the deleted position.



Since item 11 in its original position conforms to rule (0), nothing else needs to be done. So, the whole process is completed.

4.

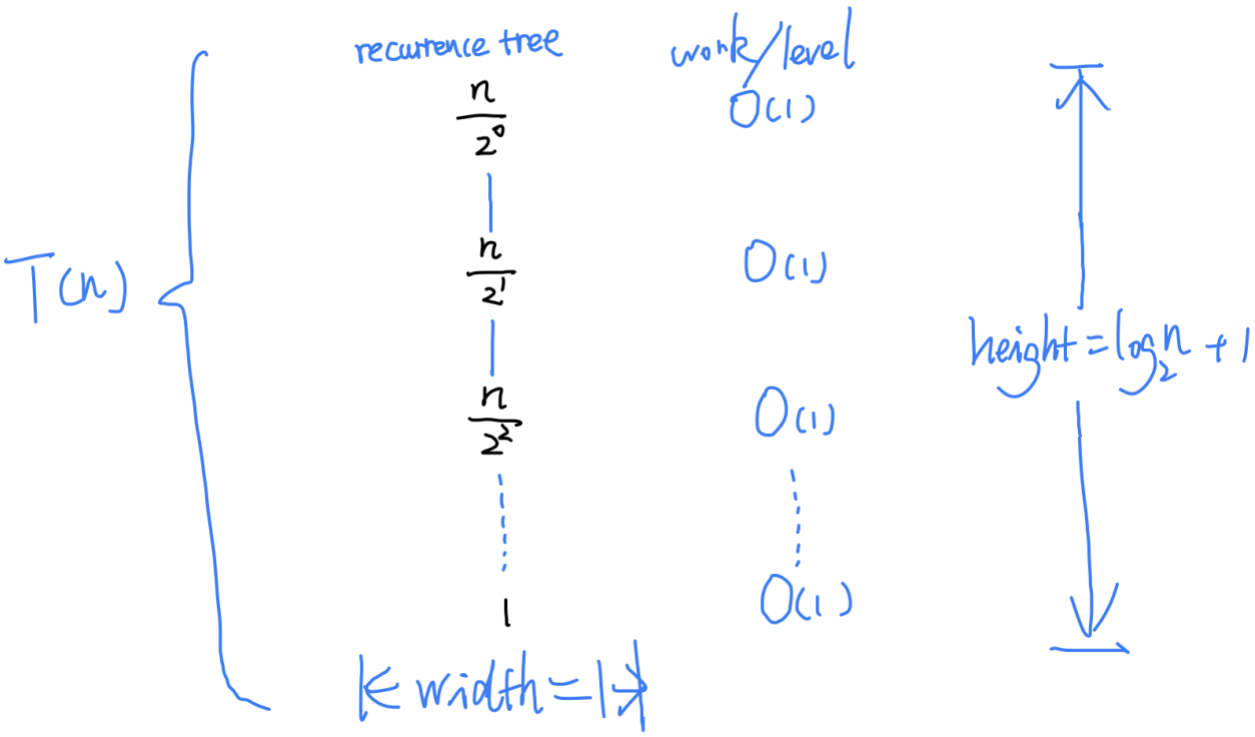
When searching in a binary search tree, we start from the root and ignore approximately half a tree every time traversing to the next child node. And when we traverse to a node, what we do is only compare the data on the node and the data we are searching for. Therefore, we can have:

$$T(n) = 1 * T(n/2) + O(1)$$

Where:
a = 1
b = 2
f(n) = O(1)

5.

According to question 4:



6.

According to the master theorem (floor/ceiling notations ignored), if:

$$T(n) = a \cdot T(\frac{n}{b}) + O(n^d) \quad for\ constants\ a > 0, b > 1, d >= 0$$

Then:

$$T(n) \in \begin{cases} O(n^d) & if\ d > log_b^a \\ O(n^d \cdot log_2^n) & if\ d = log_b^a \\ O(n^{log_b^a}) & if\ d < log_b^a \end{cases}$$

According to question 4, our equation is:

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + O(1) = 1 \cdot T\left(\frac{n}{2}\right) + O(n^0)$$

Therefore, in our case, a = 1 > 0, b = 2 > 1, d = 0 >= 0, which conforms to the basic condition. Moreover:

$$d = 0$$
$$log_b^a = log_2^1 = 0$$

Since they are equal, we can have:

$$T(n) \in O(n^d \cdot log_2^n) = O(n^0 \cdot log_2^n) = O(log_2^n)$$