Tutorial (week 7)

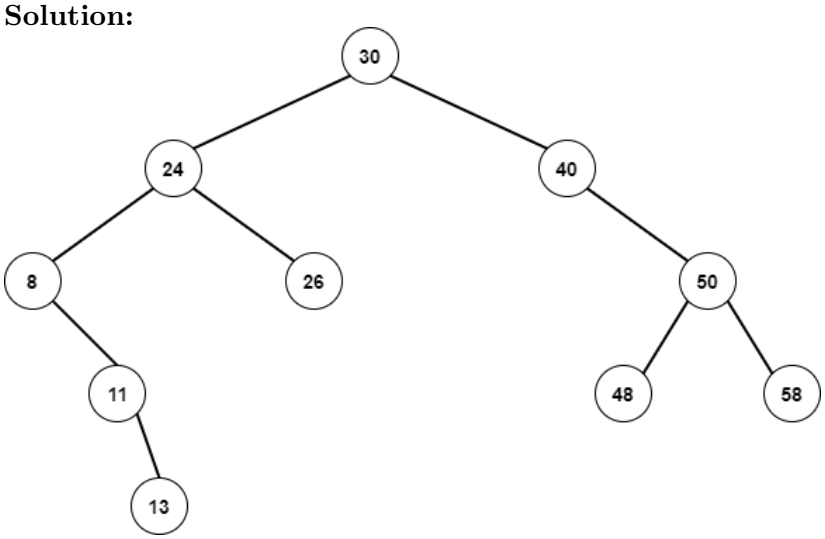
All the answers can be given with pseudocode or with Python. Remember that most of the time, many solutions are possible.

1. Suppose you are given the array $A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$, and you then perform a binary search algorithm to find the number 8. Which numbers in the array A are compared against the number 8 ?

   Solution:
   Considering that the middle item is computed as $mid = \lfloor (low + high)/2 \rfloor$, the sequence is 6 - 9 - 7 - 8.

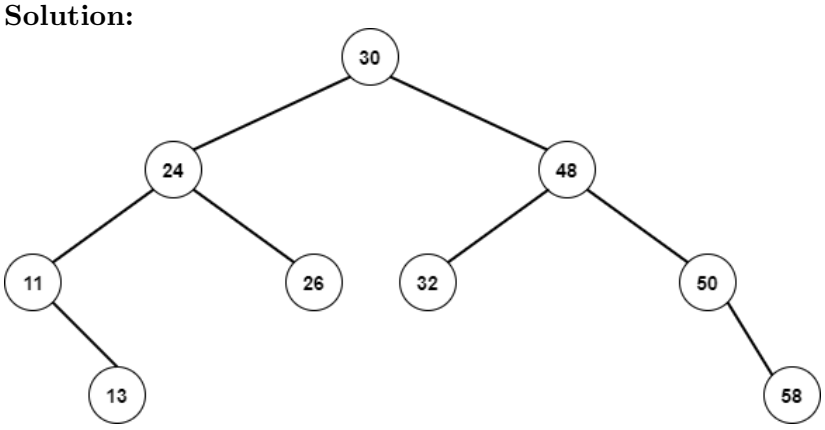
2. Insert items with the following keys (in the given order) into an initially empty binary search tree: 30, 40, 50, 24, 8, 58, 48, 26, 11, 13. Draw the tree that results.

   Solution:



3. From the final tree of the previous question, add the element 32 and then remove the elements 8 and 40. Note that two answers are possible, depending on how you select the node to reinsert.

   Solution:



4. Suppose you have a binary search tree, T, storing numbers in the range from 1 to 500, and you do a search for the integer 250. Which of the following sequences are possible sequences of numbers that were encountered in this search. For the ones that are possible, draw the search path, and, for the ones that are impossible, say why.

   (a) (2, 276, 264, 270, 250)

   (b) (100, 285, 156, 203, 275, 250)

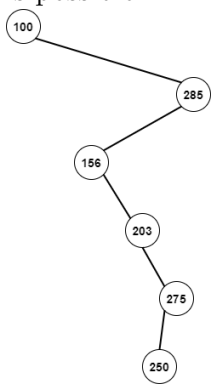
   (c) (475, 360, 248, 249, 251, 250)

   (d) (450, 262, 248, 249, 270, 250)

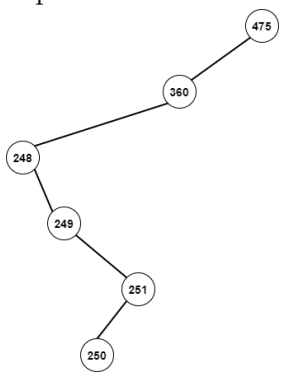
   Solution:

   (a) is impossible since when reaching 264, you will choose the left subtree of 264 since $250 < 264$. It is therefore impossible that you later encounter 270 which is bigger than 264.

   (b) is possible.



   (c) is possible.

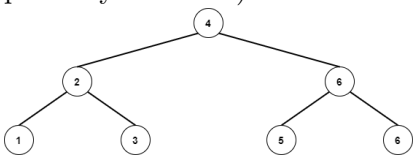


   (d) is impossible since when reaching 262, you will choose the left subtree of 262 that corresponds to numbers smaller than 262. It is therefore impossible that you later encounter 270 which is bigger than 262.

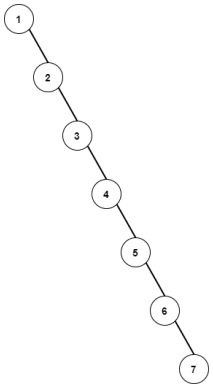
5. Draw the binary search trees of minimum and maximum heights that store all the integers in the range from 1 to 7, inclusive.

   Solution:
   The minimum case happens when the tree is as balanced as possible (in this case, we can make it

perfectly balanced).



The maximal case happens when the tree is completely unbalanced, basically just a straight line.



6. Give a pseudocode description of an algorithm to find the element with smallest key in a binary search tree. What is the running time of your method ?

   Solution:
   The algorithm is very simple: just start from the root of the tree and keep choosing the left subtree until you reach an empty node. The complexity is $O(h)$, where h is the height of the tree. If the tree is balanced, this is $O(\log n)$.

7. A certain Professor Amongus claims that the order in which a fixed set of elements is inserted into a binary search tree does not matter (the same tree results every time). Give a small example that proves Professor Amongus wrong.

   Solution:
   Draw the binary search tree created by the input sequence: 1, 2, 3 (you get a fully unbalanced tree). Now draw the tree created when you insert the input sequence: 2, 3, 1 (you get a perfectly balanced tree).

8. Suppose that a binary search tree, T, is constructed by inserting the integers from 1 to n in this order. Give a big-Oh characterization of the number of comparisons that were done to construct T.

   Solution:
   When inserting the integers from 1 to n in this order, you will obtain a fully unbalanced tree, with 1 as root of the tree and where all integers are placed as right children of the previous integer.

   When inserting 1, you will make 0 comparison as the tree was empty. When inserting 2, you will make one comparison with 1, then insert the node on its right. When inserting 3, you will make one comparison with 1, one comparison with 2, then insert the node on its right, etc. Finally, when inserting n, you will make $n-1$ comparisons. Overall, the number of comparisons is

$$\sum_{i=1}^{n-1} i = n(n-1)/2$$

which is $O(n^2)$.

9. Suppose you are given a sorted array, A, of n distinct integers in the range from 0 to n, so there is exactly one integer in this range missing from A. Describe an $O(\log n)$-time algorithm for finding the integer in this range that is not in A.

Solution:
Use a binary search to look for the missing element.

```
import math

def missing(A):
    '''
    INPUT: A sorted array A of positive integers
    where all integers in [0,n] are present except one
    OUTPUT: the missing integer x
    '''

    # we have one low and one high pointer
    low = 0
    high = len(A)-1

    # if all first n-1 elements are in proper position, then it is n that is missing
    if A[high] == high:
        return high+1

    # we do a binary search for the missing element, updating low and high
    while True:
        if low == high:
            return low

        mid = math.floor((low + high)/2)

        if mid == A[mid]:  # the missing element is in the right part of the subarray
            low = mid+1
        else:              # the missing element is in the left part of the subarray
            high = mid


A = [1,2,3,4,5,6,7,8,9,10]
print(missing(A))
```

10. Describe how to perform the operation `findAllElements`(k), which returns every element with a key equal to k (allowing for duplicates) in an ordered set of n key value pairs stored in an ordered array, and show that it runs in time $O(\log n + s)$, where s is the number of elements returned.

Solution:
First, use a binary search to look for one element X with key k in the ordered array (this costs $O(\log n)$). Then, since all other elements with the same key must be consecutive, we check forward and backward from X if other elements with key k do exist (this costs in the worst case $2s$ operations, thus $O(s)$). In total, the algorithm cost is $O(\log n + s)$.

```
import math

def findAllElements(A,k):
    '''
```

```
    INPUT: A sorted array A and a key k
    OUTPUT: the indexes of all the elements of A equal to k
    '''

    # we have one low and one high pointer
    low = 0
    high = len(A)-1
    indexes = []

    # we do a binary search for an element with key k, updating low and high
    while True:
        if low == high:
            mid = low
            break

        mid = math.floor((low + high)/2)

        if A[mid] == k:    # we found an element with key k
            break
        elif A[mid] < k:   # elements with key k should be in the right half-array
            low = mid+1
        else:              # elements with key k should be in the left half-array
            high = mid

    # we go backward from mid to find the first element with key k
    temp = mid
    while A[temp-1] == k:
        temp = temp-1
        if temp == 0:
            break

    # from that first element, we go forward to find all the elements with key k
    while A[temp] == k:
        indexes.append(temp)
        temp = temp+1
        if temp == len(A):
            break

    return indexes

A = [1,2,3,4,4,4,5]
print(findAllElements(A,4))
```

11. Describe how to perform the operation `findAllElements`(k), as defined in the previous exercise, in an ordered set of key-value pairs implemented with a binary search tree T, and show that it runs in time $O(h+s)$, where h is the height of T and s is the number of items returned.

    Solution:
    One possible solution: we find all the elements with key k recursively, using a classical binary search. You can view this as a range query in T with $k_1 = k_2 = k$, which is $O(h+s)$ as seen during the lecture.

Hints

- **Question 1:** be sure to keep the low, mid, and high variables straight during the algorithm simulation.
- **Question 2:** Review the algorithm for inserting items in a binary search tree.

- **Question 4:** Review the definition of a binary search tree.
- **Question 5:** Review the definition of a binary search tree.
- **Question 6:** Review the selection algorithm given in the book, and specialize it for the case $i = 1$.
- **Question 8:** Recall the worst-case example of the height of a binary search tree.
- **Question 9:** Think about how to devise a probe test to determine if you are querying above or below the missing number.