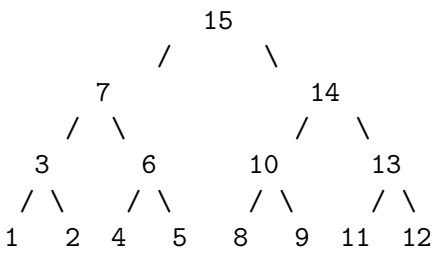
1. (a) $T(1) = 3$, $T(n) = T(n/2) + 3$ for $n > 1$.

   (b) Use the Master Theorem, with $a = 3$ and $b = 3$, so $E = log3/log3 = 1$. $f(n) \in \Theta(n)$ holds, so this is case 2, so $T(n) \in \Theta(nlog(n))$.

2. (a) Yes. In the lecture we saw algorithms (buildHeap and constructHeap) that transform any array (so also a minheap) into a maxheap, with complexity $\Theta(n)$.

   (b) The tree is



   The sequence: 15, 7, 3, 1, 2, 6, 4, 5, 14, 10, 8, 9, 13, 11, 12.

   (c) We search for the biggest element in the BST by going to the right as much as possible and so arrive at node x. Then we search for the biggest element smaller than $x.key$: if x has a left child we first go to the left, and then as much to the right as possible. If x has no left child we take the parent of x.

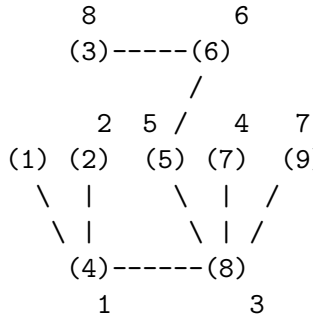
```
def next_to_biggest(T):
    x=T.root

    while x.right != null:
        x=x.right

    if x.left == null:
        return x.parent
    else:
        x=x.left
        while x.right != null:
            x=x.right
        return x
```

3. • Base case: for $h = 0$ there can be at most 1 node, which is a leaf node, so no internal nodes, so $2^h - 1 = 0$, so it holds.

- Induction step: Suppose it holds for all i with $0 \leq i \leq h$. Now a tree of height $h+1$ consists of a root node (which is an internal node), and a left and right subtree with heights smaller or equal to h, so according to the induction hypothesis they each have at most $2^h - 1$ internal nodes. So total amount of internal nodes maximally $1 + (2^h - 1) + (2^h - 1) = 2^{h+1} - 1$, so the property also holds for $h+1$.

So, according to the principle of mathematical induction, the property holds for all $h \geq 0$.

4.



5. (a)
   - Either you do not include integer a_i in the sum, then the possibility of forming the sum g with the remaining numbers is given by $R(i-1, g)$
   - or you do include a_i in the sum, and now the problem is whether it is possible to form the sum $g - a_i$ with the remaining numbers, which is given by $R(i-1, g-a_i)$
   - now $R(i, g)$ is the "or" of these two possibilities

   So the correct option is (ii): $R(i, g) = R(i-1, g) || R(i-1, g-a_i)$

   (b) The algorithm to fill the boolean matrix `R` (we use the indices in `a` ranging from 1 to n, so we do not use element $a[0]$):

```
def subsum(a,G):

    n=len(a)-1
    R=[[0 for g in range(G+1)] for i in range(n+1)]

    for i in range(0,n+1):
        R[i,0]=1      #always possible to form sum 0

    for i in range(1,n+1):
        for g in range(1,G+1):
            if (g-a[i])<0:
                R[i,g]=R[i-1,g]
            else:
                 R[i,g]=R[i-1,g]||R[i-1,g-a[i]])

    return R[n,G];
```

The complexity of this algorithm is $\Theta(nG)$.