Prob. 1	Prob. 2	Prob. 3

Problem 1.

1. In the worst case, we insert the smallest number into tree, so we need to swap from the leaf to the root in *bubble-up* way. That results in $\lfloor \log(n) \rfloor$. In the worst case after the delete, we replace the root with the largest value then we need to swap from the root to the leaf in *bubble-down* way. That results in $\lfloor \log(n) \rfloor$.

2. For amortized analysis, we can sum depths of all nodes as a potential function. Therefore, since tree is balanced, we sum *logarithms* of all nodes and we have the following for n items of a heap:

   $$\Phi = \sum_{i=2}^{n} \log i$$

   insert operation has a logarithmic amortized cost:

   $$\textit{amortized cost} = \textit{actual cost} + \Delta\Phi = \log(n) + \log(n+1) \equiv O(\log n)$$

   However, *delete* operation has better amortized cost:

   $$\textit{amortized cost} = \textit{actual cost} + \Delta\Phi = \log(n) - \log(n) \equiv O(1)$$

Problem 2.

1. As we have seen in the first question, the depth of a *balanced* tree is $\log n$. If we have a node that has a *depth* larger than $b \cdot \log n$, this is a sign of a unbalanced tree because b is larger than 1 since c is in the range of $({}^{1}/_{2}, 1)$. Therefore, in the path from x to *root*, we will find a point where that unbalance comes from. Since tree is binary, if tree was balanced, each child would equally share the nodes of its parent, that means each child can have at most half of weight of its parent. However, it is unbalanced so one child will ceratinly have nodes under its subtrees more than half of weight of its parent.

2. Since we rearrange the tree to make it balanced again when a node has a degree larger than $b \cdot \log n$, the height of tree will be bounded by the bound we do the rearrangement, which is $b \cdot \log n$, here.

3. - *find* operation can take the longest path in the worst case such as unsuccessful search. From point 2, we know that the height of tree is bounded by $b \cdot \log n$ so the longest path is bounded by $O(\log n)$.
   - *add* operation is more complex compared with *find*. Its cost is composed of executing find and then checking weights from the node, where the *find* took us, to root (here we assume that each node knows to answer its weight) and doing rearrangement. In the worst case, we will rearrange the whole tree so we have the following n items of tree and T denotes the whole tree:

$$\begin{aligned} add &= find + weight\ check + rearrangement \\ &= O(\log n) + O(\log n) + sort(T) + create(T) \\ &= O(\log n) + O(n \log n) + (2T({}^{n}/_{2}) + O(1)) \\ &= O(n \log n) + O(n) \\ &= O(n \log n) \end{aligned}$$

4.

Problem 3.