Q1: Argue that since sorting n elements takes O(n lg n) time in the worst case in the comparison model, any comparison based algorithm for constructing a binary search tree from an arbitrary list of n elements take O(n lg n) time in the worst case.

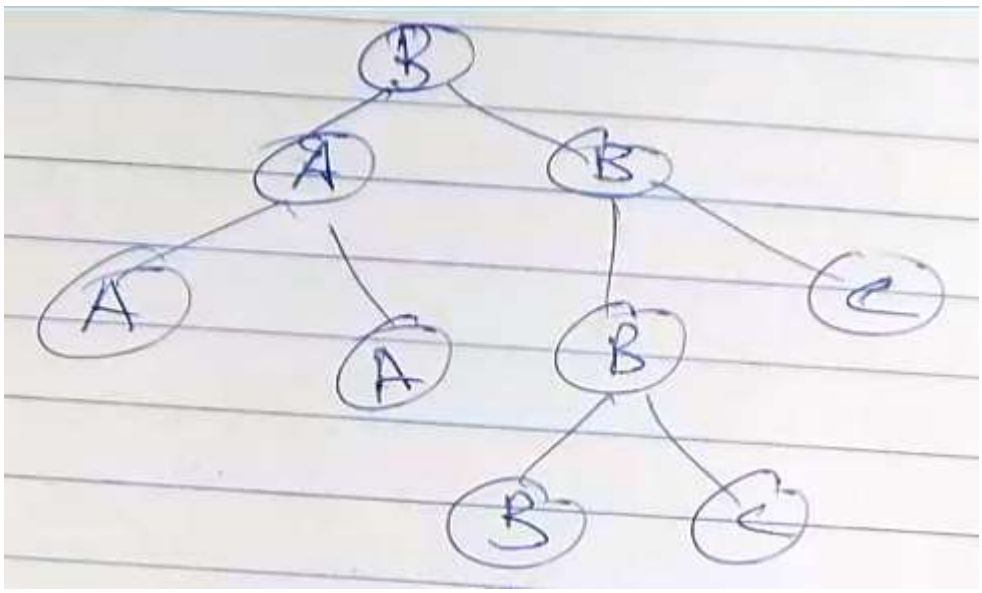
A: If it is not O(nlgn), we will find a conflict.

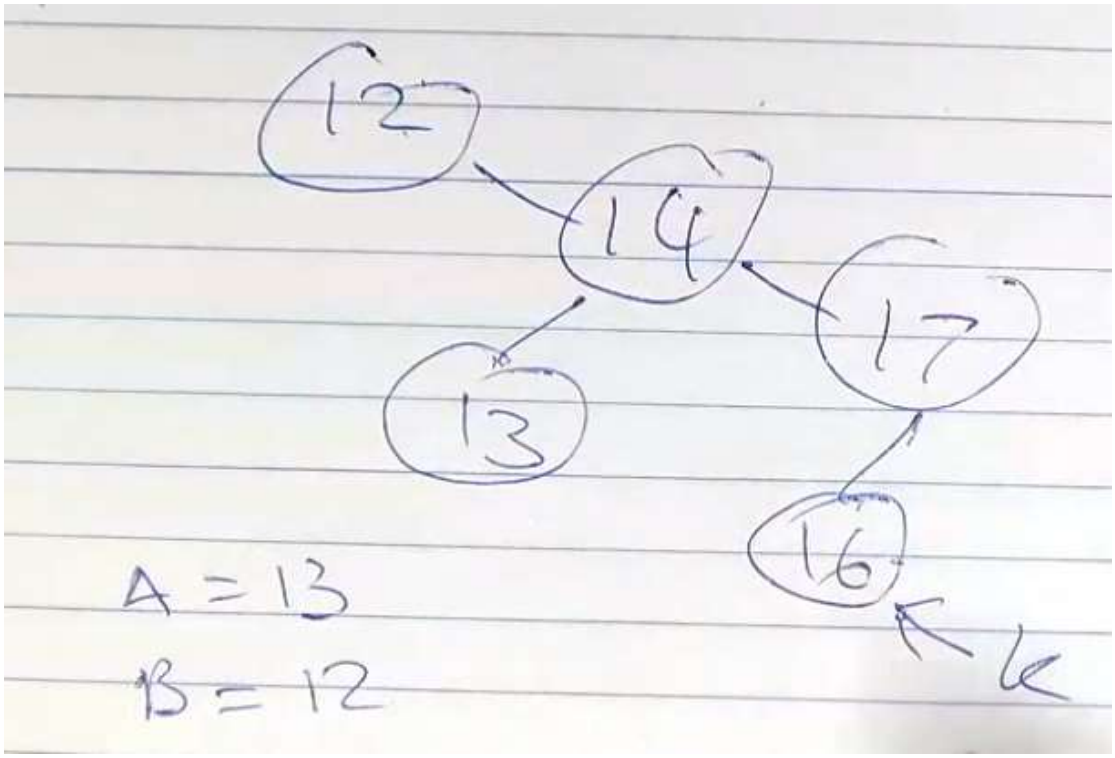
Given n unsorted items, to sort, we would just construct the binary search tree and then read off the elements in an in-order traversal, which will give us a sorted list. This second step can be done in time O(n) as we learned from the lecture.

Also, an in-order traversal must be in sorted order because the elements in the left subtree are all those that are smaller than the current element, and they all get printed out before the current element, and the elements of the right subtree are all those elements that are larger and they get printed out after the current element.

Now this question restricts us to construct the tree using any comparison-based algorithm. So If the lower bound in the worst case in the comparison model is better than O(nlgn), then we can utilize this to give a better comparison sorting algorithm. This conflicts with that sorting n elements takes O(n lg n) time in the worst case in the comparison model.

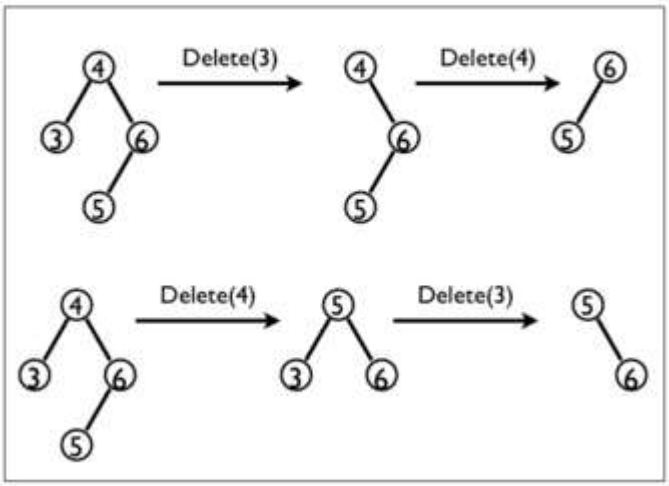
Q2: Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leqslant b \leqslant c$. Give a smallest possible counterexample to the professor's claim.





Q3: Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

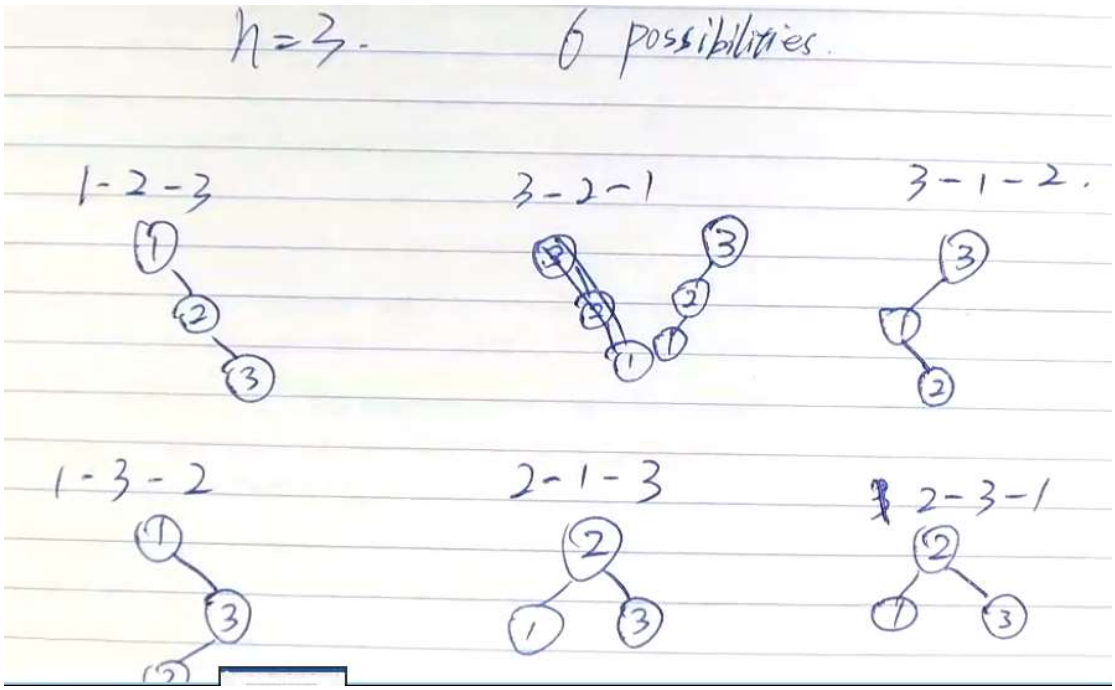
Answer



NO.

Q4: Show that the notion of a randomly chosen binary search tree on n keys, where each binary search tree of n keys is equally likely to be chosen, is different from the notion of a randomly built binary search tree given in this section. (hint: list the possibilities when n=3)

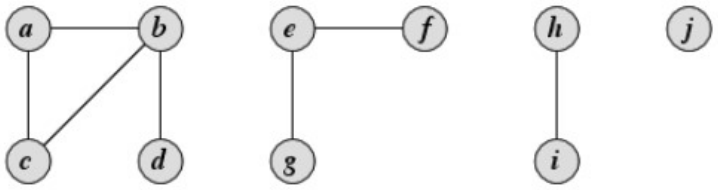
A: Suppose we have the elements {1, 2, 3}. Then, if we construct a tree by a random ordering, then, we get trees which appear with probabilities some multiple of 1/6 . However, if we consider all the valid binary search trees on the key set of {1, 2, 3}. Then, we will have only five different possibilities. So, each will occur with probability 1/5 , which is a different probability distribution.



if we consider all the valid binary search trees on the key set of {1, 2, 3}. Then, we will have only five different possibilities, Last two are same cases.

Q5: Write pseudocode for MAKE-SET, FIND-SET, and UNION using the linked-list representation and the weighted-union heuristic. Assume that each object x has an attribute rep[x] pointing to the representative of the set containing x and that each set S has attributes head[S], tail[S], and size[S] (which equals the length of the list).

Disjoint set



(a)

Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

This is my graph, and of course not all the graphs are connected together. Then we want to figure out how many groups are there. So if two nodes are connected, even they are not connected directly, we say they are in the same group. We basically want to find out how many groups are there, the algorithm is called disjoint union. You need have sort of different operations.

MAKE-SET(x) creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

UNION(x, y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets S_x and S_y, removing them from the collection $\mathcal{S}$. In practice, we often absorb the elements of one of the sets into the other set.

FIND-SET(x) returns a pointer to the representative of the (unique) set containing x.

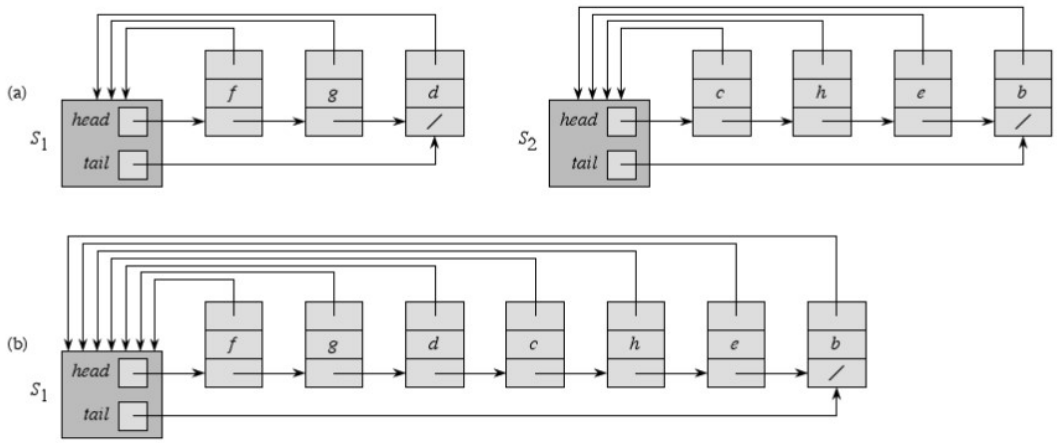


Figure 21.2 **(a)** Linked-list representations of two sets. Set S_1 contains members d, f, and g, with representative f, and set S_2 contains members b, c, e, and h, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers *head* and *tail* to the first and last objects, respectively. **(b)** The result of UNION(g, e), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f. The set object for e's list, S_2, is destroyed.

Answer

```
MAKE-SET(x):
        Initialize a new linked list
        Insert node x

FIND-SET(x):
        return rep[x]

UNION(x, y):
        Let smallist,biglist be the list with less and larger list according to size[x],size[y]
        put smallist to the tail in biglist
```