1. (2 points) I have a binary tree (**not** a binary search tree) with the following properties:

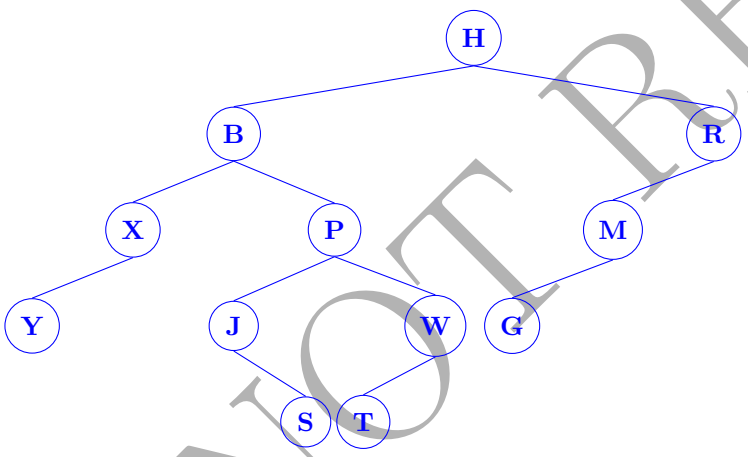
   - Each non-null node of T contains a single character
   - An in-order traversal of the tree reads "YXBJSPWTHGMR"
   - A post-order traversal of the tree reads "YXSJTWPBGMRH"
     This is a reminder that this is not a binary search tree.

   Give a pre-order traversal of the tree.

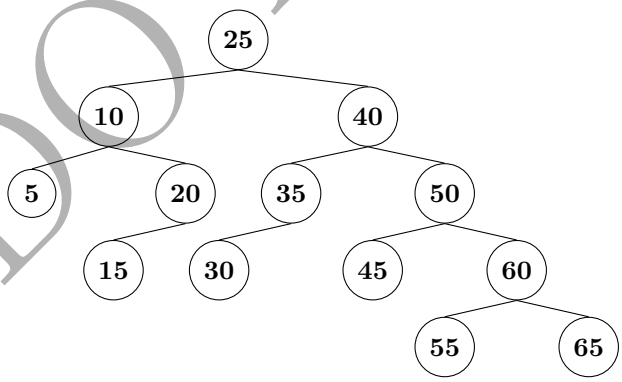
   We derive this similar to the homework problem. H is last in the post-order traversal, so it appears at the root. The three elements after it in the in-order traversal constitute its right subtree; R is the last of these in the post-, so it must be the root of the right subtree; because it is last in-order, the other two must appear in its left-subtree.

   For the left sub-tree of the root, we recursively solve the problem of having an in-order traversal of YXBJSPWT and a post-order of YXSJTWPB. We derive that B is the root of the left sub-tree, with YX to its left and JSPWT to its right.

   Here is the tree. The pre-order traversal is HBXYPJSWTRMG



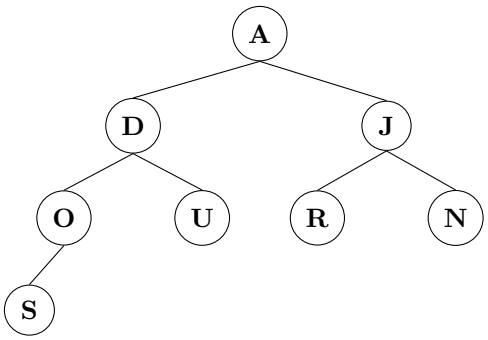
2. (1 points) Consider the following AVL Tree. Which key can I delete in the following tree to cause two separate re-balancing operations?



   Deleting 5 causes 10 to be imbalanced. The re-balance causes 25 to become such. 30 or 35 cause one rotation, not two. Deleting 65 causes none

3. (1.5 point) In lecture, we saw that an array can be considered a complete tree and, if the heap property applies, it is also a heap.

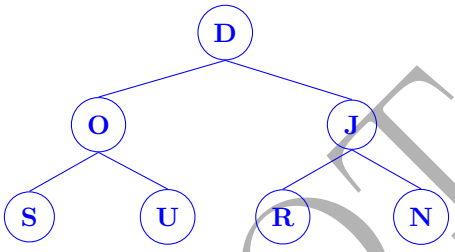
   (a) Here is a binary heap, drawn as a tree. What is the character array/string representation of the heap? Write only the eight characters in the form.



   ADJOURNS – yet another word that is a heap; your professor is amused by this sort of thing.

   (b) What is the result of a `remove-min` operation on this heap? You may want to draw the resulting heap, and on the answer page, you will describe the result.

   After making the change, the heap looks like:



   (c) What is the string/vector representation of the resulting heap?
   The answer is "DOJSURN" (be sure you write only those seven characters in the form)

4. (2 points) Consider the following Binary Search Tree, being balanced via the rules of AVL-Trees. We are examining it in the middle of the "insert" operation. A Key has been inserted into the tree, but we have not (yet) performed any rotations.

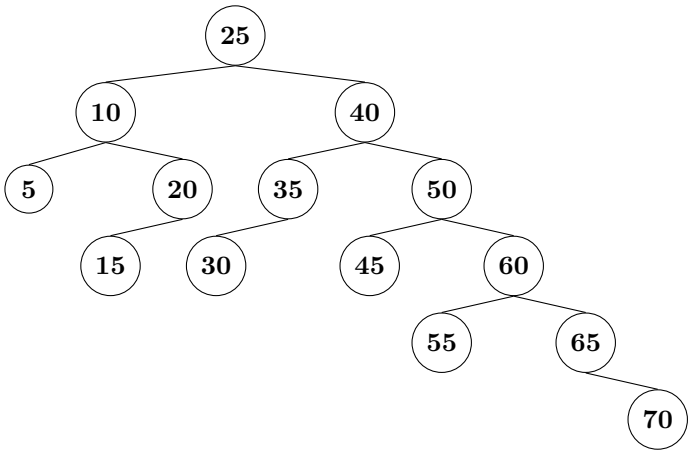
   (a) Which key did we just insert?

   (b) In setting up the update operation, what is the key stored at z, as described in lecture?

   (c) In setting up the update operation, what is the key stored at y, as described in lecture?

   (d) In setting up the update operation, what is the key stored at x, as described in lecture?

   We just inserted 70. $z = 50$ is the point of re-balance, with 60 and 65 and y, x respectively.



5. (2 points) In project 4, you did not have to write a delete on your AVL tree. You're going to write it here.

   You may assume every node in the tree stores a Key,Value pair, along with left, right, and parent pointers. You also have the following helper functions already written and tested (you may call them and assume they work):

   - `Node * find(Key k)` returns a pointer to a Node that has the given key. It returns `nullptr` if the key is not in the tree.
   - `Node * rebalance(Node * z)`, which performs the local rebalancing, given that z is the lowest unbalanced node. This is the same definition of z we used in lecture.
   - `Node * localDelete(Node * n)`, which deletes the given node, freeing all relevant memory. It returns a pointer to the "replacement" (same child of same parent) if a non-leaf with one child is deleted, the parent if a leaf is deleted, or the actually-deleted node if the parameter had two children (after moving the key/values appropriately). This is the "normal BST" localDelete, and does not do any height re-balancing.
   - `int height(Node *n)`, which returns the height of the given node. That is, the number of hops to fall off the farthest leaf underneath it. If the parameter is `nullptr`, it returns -1.

   While we will not require strict C++, you should write your code in such a way as to make it very clear to the grader that you *could* implement it correctly from your ideas.

   `void delete(Key k)`

   First, find the Node we're going to delete. If it isn't there, do nothing (catch exception and return).

   If it is there, call localDelete() on it and retain the returned pointer.

   Walk up from that pointer. At each node on the way up, check if it is balanced: if abs(height n's left, height of n's right) > 1, rebalance at that node and set the "cur" pointer to the node.

   Note that delete might cause 2+ rebalancing operations, so you can't just stop at one like you can on insert!

6. (2 points) Suppose we are writing code for a priority queue, implemented as a min heap (using a `std::vector`). We have `insert, min` and `extract-min` written and we want to add a function `where-is-key`(k). This function is designed to answer the following query: is k in the heap, and if so, which index in the vector holds k? I want this to run in $\mathcal{O}(\log n)$ time. The other operations' running times may increase by a factor of $\mathcal{O}(\log n)$ if needed. Explain how to do this. You may use $\mathcal{O}(n)$ additional space to support this operation.

   You will still get full credit if your running time is *in expectation* $\mathcal{O}(\log n)$ instead of guaranteed, and/or your $\mathcal{O}(n)$ additional space is in expectation, but if either or both is in expectation, you need to state that.

   After you take the diagnostic test, and before you read the answer to this question, discuss this question and your solutions with your study group for a few minutes. Of course, make sure they have taken the diagnostic test too!

   Solution with $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ space guaranteed: when we create the PQ, make also an AVL Tree. The key for the AVL Tree is the keys in the heap, and the value is an integer index at which this key lives in the priority queue. Whenever we insert an element into the heap, add it to the AVL Tree with a value equal to its starting index. When we swap two elements in the heap, swap their values (this takes $\mathcal{O}(\log n)$ due to the find operation, and happens at every swap, potentially $\mathcal{O}(\log n)$ of them, for $\mathcal{O}(\log^2 n)$ time for insert into the heap) in the tree.

   We do similarly on extract: delete the previous root from the AVL Tree, change the value of the element that got moved to the root's position, and update on future swaps for $\mathcal{O}(\log^2 n)$ time for these. We do not need to update the tree when we call function min. In order to support `where-is-key`, we simply look up the key in the AVL Tree.

   Alternate solutions:

   (a) Use a regular BST instead of an AVL Tree. This works as above, and now has the time in expectation but space is still guaranteed.

   (b) Use a skip list instead of an AVL Tree. The time and space are now in expectation.