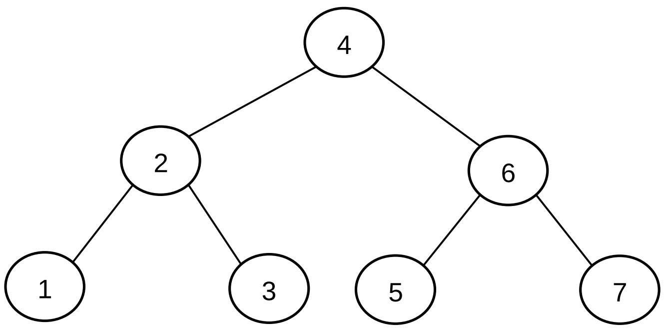
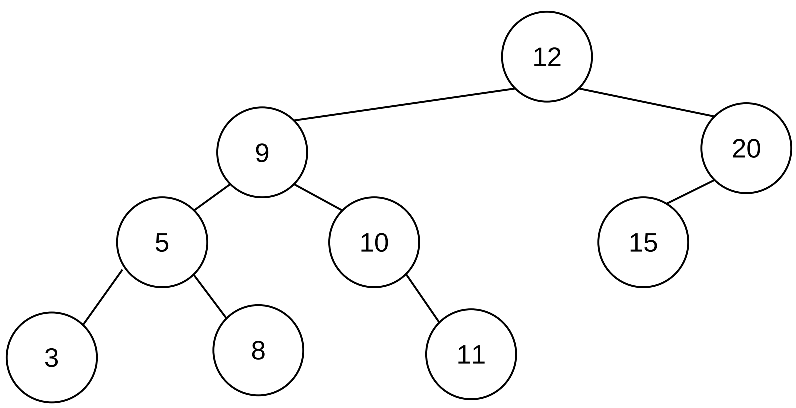
Section 4 SOLUTION: AVL Trees & B-Trees

1. **What 3 properties must an AVL tree have?**
   a. Be a binary tree
   b. Have Binary Search Tree ordering property (left children < parent, right children > parent)
   c. Be a balanced tree: | n.left.height – n.right.height | ≤ 1 for all nodes n in the tree.

2. **In a typical BST, inserting keys in order result in a worst-case height. Show the result when an initially empty AVL tree has keys 1 through 7 inserted in order.**



3. **AVL tree balance violation cases:**
   a. **Insert the following keys, in order, into an initially empty AVL tree**: 12, 8, 9, 20, 10, 15, 3, 11, 5.



   b. **Find a key we could insert into your resulting tree that would result in a case 1 balance violation (left-left).**

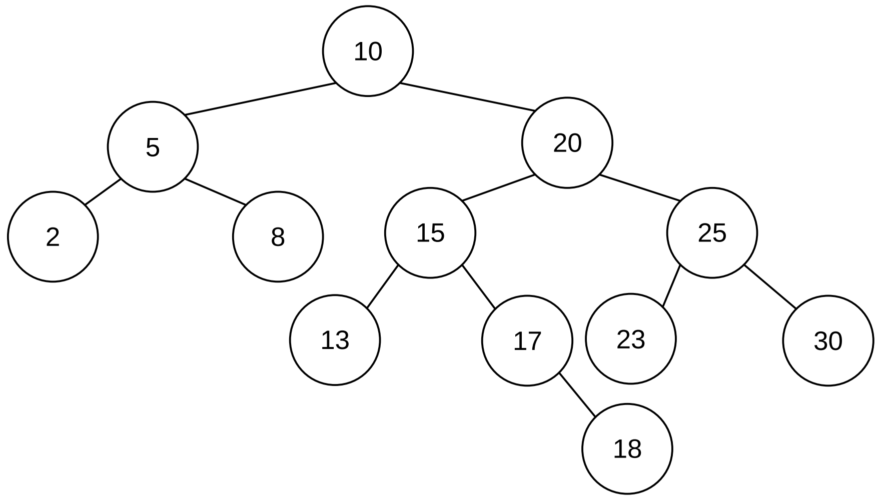
   There are several keys we could insert to get a case 1 rotation; inserting ‘1’, for instance, will cause a height imbalance to be detected at the root.

4. **For the following AVL tree:**

   a. **What values could you insert to cause a right-right imbalance, and at which node does the imbalance occur?**
   Inserting any value greater than 20 causes a right-right imbalance at node 10, the root, because the root’s left child now has height 1 and the right child has height 3.

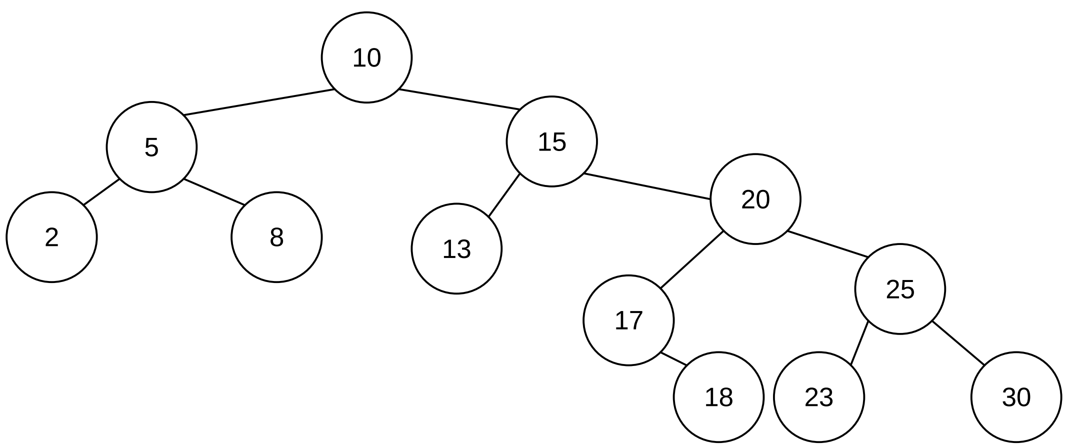
b. **How about a right-left imbalance? At which node does the imbalance occur?**
Inserting any value GREATER than 10 and LESS than 20 causes a right-left imbalance also at the root, for the same reason.

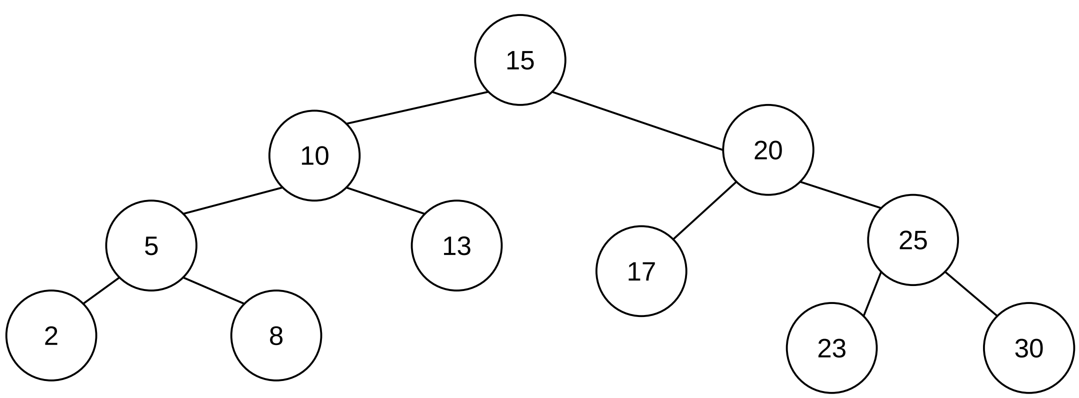
c. **Insert 18 into the following AVL tree. What type of imbalance does it cause? Show the result after balancing.**
Inserting 18 causes a right-left imbalance at the root node 10, because its left child is of height 1



and its right child is of height 3, meaning the difference is 2, which is greater than the allowed difference of 1.

To fix this node, you need to do a double rotation. First, rotate grandchild of 10 with child of 10,





that is, rotate 15 with 20 clockwise, making sure to re-assign 13, subtree starting at 17, and subtree starting at 25 correctly.
Now do your second rotation, rotate the node counterclockwise where the imbalance was found (node 10) with its imbalance inducing child 15.

5. **Given a binary search tree, describe how you could convert it into an AVL tree with worst-case time O(nlogn) and best case O(n).**
   We can traverse the tree in O(n) time and insert each element into an initially empty AVL tree; this will take O(nlogn) time overall. To get O(n) 'best-case' performance we can do something that's a bit of a hack: try to verify that the BST is a valid AVL tree, which we can do in O(n) time; if it is, return it as it is. Otherwise, create a new AVL tree as described. It's questionable as to whether this is really a 'best-case' result, as we don't actually do any conversion, only verification.

6. **Say you work for a software company, and your boss wants you to implement priority-queue-like functionality for a project you are working on, and he suggests you just use the AVL tree you already have implemented, since you can findMin/insert/deleteMin all run in O(logn) time guaranteed anyway.**
   a. **Give a couple of reasons why you may want to go with a binary min heap instead.**
      Some reasons:
      i. Since we're implementing a priority queue, finding a min on a heap is always O(1), whereas finding the min on a AVL tree is O(logn).
      ii. Although they have a O(logn) insert as well, binary heaps have an *average* O(1) expected time for inserts, since ½ of the elements are leaves, and most elements are in the bottom two levels.
      iii. We'll likely get better caching performance from the binary heap, as it's stored as an array
      iv. AVL tree pointers take up additional memory

   b. **Regarding the situation in part (a), under what additional circumstances may the AVL be preferable?**

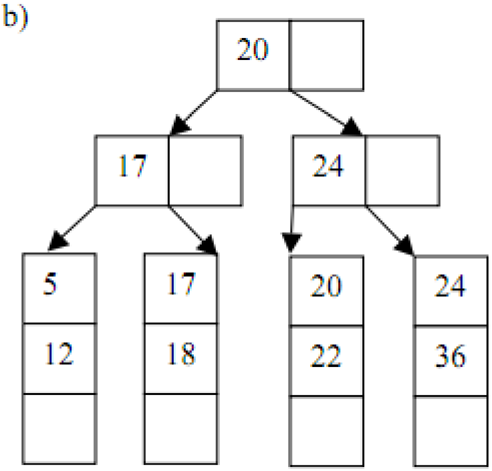
      If we want to perform other operations, like finding a patient record based on their name rather than their priority, etc., an AVL tree that used patient names as the key would be useful. Also, an AVL tree will use extra the memory it needs, whereas a (potentially large) portion of the array for a binary heap may go unused; depending on the circumstances, it's possible that the binary heap could use more memory.

7. **B-Tree**
   a. **What constraints do the following values impose on a B-Tree: M=32, L=16?**

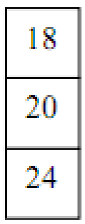
      Each node can have at most M children, and must have a minimum of M/2 children, and each leaf can have at most L data items, and at least L/2 data items. (rounding-up) So, a tree with M=32 and L=16 must have 16-32 children at each internal node, and must have 8-16 items at each leaf. (excepting the first 7 insertions

   b. **Insert the following into an empty B tree with M=3 and L=3: 12, 24, 36, 17, 18, 5, 22, 20.**

b)



c. **Delete 17, 12, 22, 5 & 36**



8. Given the following parameters for a B-tree with M= 11 and L = 8
Key Size = 10 bytes
Pointer Size = 2 bytes
Data Size = 16 bytes per record (includes the key)

Assuming that M and L were chosen appropriately, what is the likely size of a disk block on the machine where this implementation will be deployed? Give a numeric answer and a short justification based on two equations using the parameter values above.

```
Internal Node size = M * pointer + (M - 1) * key
                   = 11 * 2 + (11 - 1) * 10
                   = 22 + 100
                   = 122

Leaf Node Size = L * data
               = 8 * 16
               = 128
```

Max of these is 128 (also a power of 2) so disk block is likely to be 128 bytes