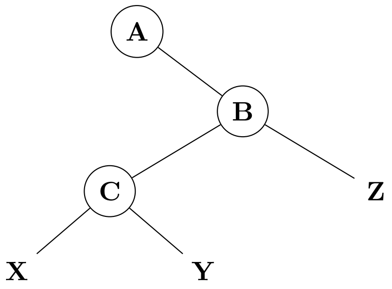
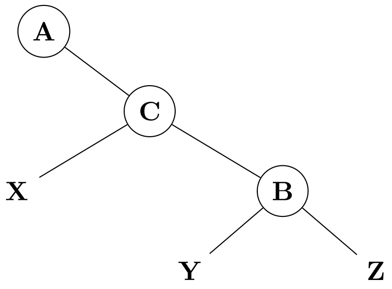
1. (*10 points*) Draw the result of the following segment of a tree after a right rotation has been done around B. The labels X, Y, and Z are to be considered subtrees, possibly empty.



Solution:



2. (*10 points*) Create a templated function that will take in one parameter, a pointer to the node that you are rotating about. It is to preform a right rotation around the parameter pointer and adjust the pointer to still point to the resulting (parent) node of the rotation.

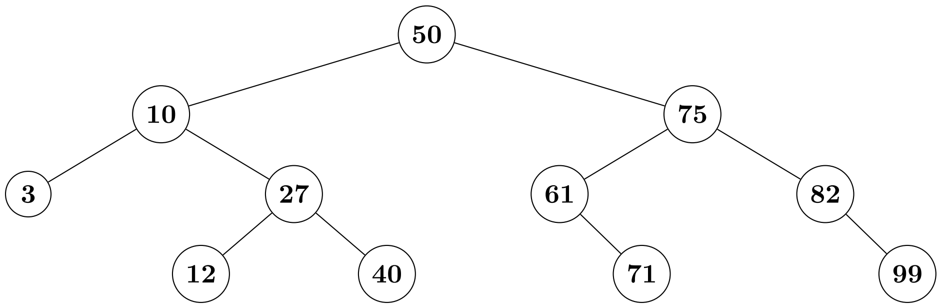
Solution:

```
template<class T>
void BinaryTree<T>::RightRotation(TreeNode *&nodePtr) {
    TreeNode *L = nodePtr->left;
    TreeNode *temp = L->right;
    L->right = nodePtr;
    nodePtr->left = temp;
    nodePtr = L;
}
```

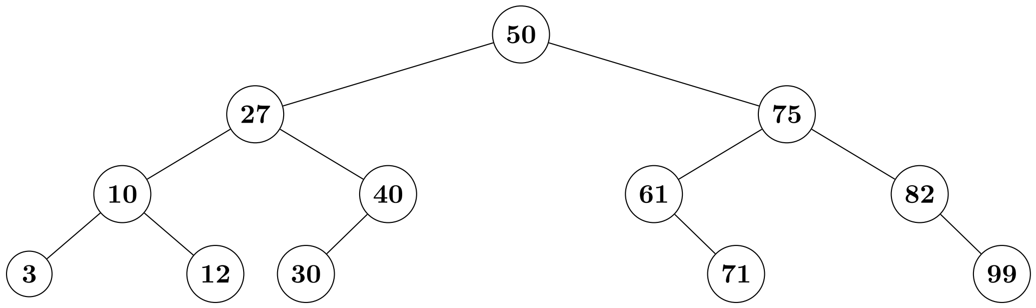
3. (*5 points*) What is the criterion for a binary search tree to be an AVL tree?

Solution: An AVL tree (originally called an admissible tree) is one in which the height of the left and right subtrees of every node differ by at most one.

4. (*10 points*) Given the following AVL tree, draw the tree after the node 30 is inserted and the tree is rebalanced.



Solution:



5. (*10 points*) What are the two phases to the DSW Algorithm? Explain what they do and how they work. Make the explanation detailed enough so that someone could reproduce the method on paper.

Solution:

(a) Create the Backbone/Vine: Convert the tree into a linked list that just has right children. This is done by starting at the root and doing right rotations until the root has no left child. Then the pointer is moved down the right branches until hitting a node with a left child. Then right rotations around this node are done until it has no left child. Then the pointer is moved down the right until hitting another node with a left child. The process continues until the movement down the right branches results in a null.

(b) Tree Rebalancing Phase: Starting at the root of the backbone, make $n - m$ left rotations from the top of the backbone. Here n is the number of nodes and $m = 2^{\lfloor \lg(n+1) \rfloor} - 1$. In words, you are rotating enough nodes to get a complete tree except for the leftover leaf nodes. Then while $m > 1$. let $m = m/2$ (integer division), do m left rotations from the root of the tree down the right branches. This will reposition the leaf nodes to the far left.

6. (*10 points*) What is the criterion for a binary search tree to be a Red-Black tree?

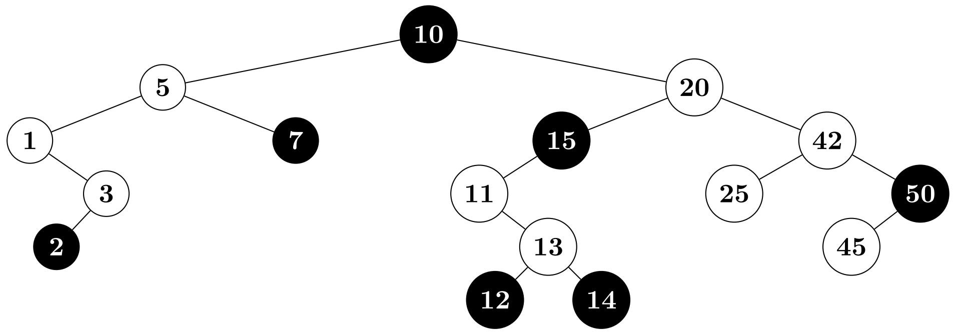
   Solution:

   A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

   A red-black tree is a binary tree that satisfies the following red-black properties:

   (a) Every node is either red or black.
   (b) The root is black.
   (c) Every leaf (NIL) is black.
   (d) If a node is red, then both its children are black.
   (e) For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

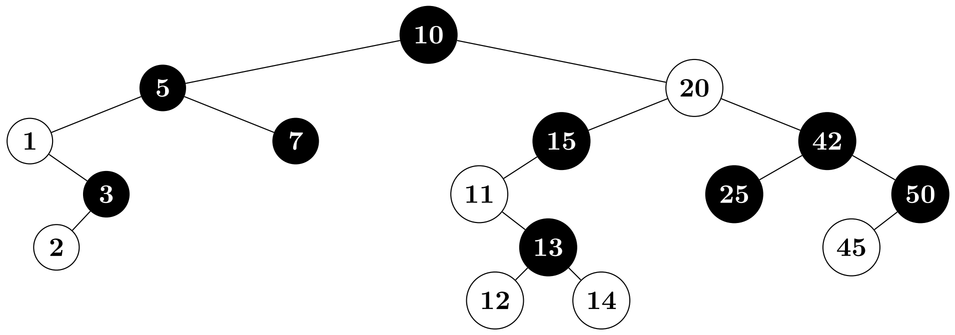
7. (*10 points*) Is the following a Red-Black Tree? If not, state all of the violations that are in the tree and then draw a fixed red-black tree. Non-filled circles are red and filled circles are black. In your fix, if you have one, shade the black nodes and do not shade the red nodes. Also, if fixing this tree, you are not to add any nodes or change the node positioning, just alter the node colors.



   Solution: This is not a Red-Black tree, the following are violations.

   (a) Nodes 5, 1, 20, 11, and 42 are red and have at least one red child.
   (b) Every path to the left of 10 has black-height 1 and on the paths through 15 the black-heights are 2. In addition, the path 10-20-42-25 has black-height 0.

   There are numerous ways to fix this, here is one solution.



8. (*10 points*) Draw the expression tree for each of the following mathematical expressions. Then rewrite the expression in postfix form.

   (a) $(x^2 + 4)/(7x - 9) + 2$

   (b) $(x + y + z)(x - yz)$

   Solution:

   (a) $(x^2 + 4)/(7x - 9) + 2$

```
   2
+
         9
      -
            X
         *
            7
   /
         4
      +
            2
         ^
            X
```

   Postfix: `X 2 ^ 4 + 7 X * 9 - / 2 +`

   (b) $(x + y + z)(x - yz)$

```
         Z
      *
         Y
   -
      X
*
      Z
   +
         Y
      +
         X
```

   Postfix: `X Y + Z + X Y Z * - *`

9. (*15 points*) Write three functions for the AVL tree.

   (a) A recursive function height that will take a pointer to a node and return the height of the subtree pointed to.

   Solution:

```
template<class T>
int BinaryTree<T>::height(Node *nodePtr) {
    if (!nodePtr)
        return 0;

    return max(height(nodePtr->left), height(nodePtr->right)) + 1;
}
```

   (b) A getBalanceFactor function will take a pointer to a node and determine the balance factor at that node. It will return the integer balance factor for the node.

   Solution:

```
template <class T>
int AVLTree<T>::getBalanceFactor(TreeNode *t) {
    if (!t)
        return 0;

    return height(t->left) - height(t->right);
}
```

   (c) The isBalanced function will take a pointer to a node and determine if the tree is AVL balanced from the node on down. This is to be a recursive function. The isBalanced function will return true if the subtree is balanced and false is not.

   Solution:

```
template <class T>
bool AVLTree<T>::isBalanced(TreeNode *t) {
    if (!t)
        return true;

    if (getBalanceFactor(t) > 1 || getBalanceFactor(t) < -1)
        return false;
    else
        return isBalanced(t->left) && isBalanced(t->right);
}
```

10. (*10 points*) Write a templated recursive function that makes a copy of a binary tree.

    Solution:

```
template <class T>
TreeNode* Tree<T>::CopyTree(TreeNode *t) {
    TreeNode *newnode;

    if (!t)
        return nullptr;

    newnode = new TreeNode();
    newnode->value = t->value;
    newnode->left = CopyTree(t->left);
    newnode->right = CopyTree(t->right);
    return newnode;
}
```