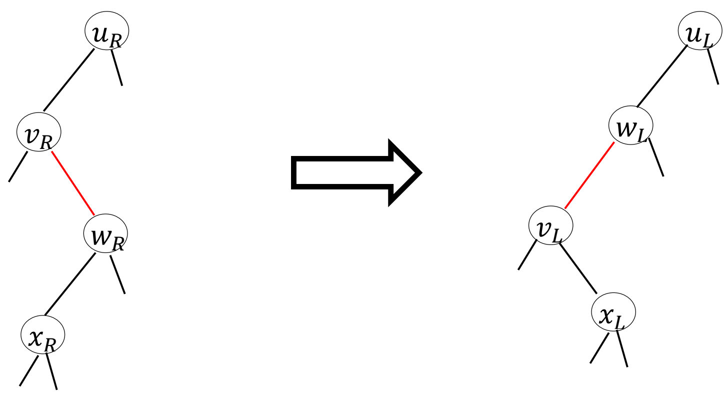
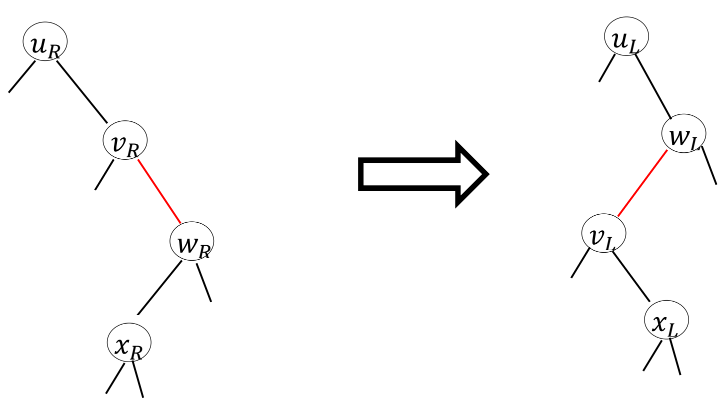
CSC 226 SUMMER 2019
ALGORITHMS AND DATA STRUCTURES II
ASSIGNMENT 2 - WRITTEN
UNIVERSITY OF VICTORIA

1. **[5 marks]** Let T_R be a right-leaning red-black tree. We will build a corresponding left-leaning red-black tree T_L as follows:

- For every node $v_R \in T_R$ with key k we create a node $v_L \in T_L$ and add key k to it. Map v_R to v_L.
- For every *red* edge $v_R w_R \in T_R$ where w_R is the right child of v_R we construct a red edge $w_L v_L \in T_L$ where v_L is the left child of w_L. Map $v_R w_R$ to $w_L v_L$.
  - Furthermore, for black edge $w_R x_R \in T_R$ where x_R is the left child of w_R, create edge $v_L x_L \in T_L$ where x_L is the right child of v_L. Map $w_R x_R$ to $v_L x_L$.
  - And, for black edge $u_R v_R \in T_R$ where u_R is the parent of v_R (whether v_R is left or right child) create edge $u_L w_L \in T_L$ with u_L the parent of w_L (where w_L is either the left or right child, corresponding to v_R's orientation in T_R.) Map $u_R v_R$ to $u_L w_L$. Visually, the two cases are shown below.



or



- For all other edges $u_R v_R \in T_R$ construct the corresponding edge $u_L v_L \in T_L$ with the same orientation. Map $u_R v_R$ to $u_L v_L$.

Now, all right-leaning red edges in T_R are left-leaning red edges in T_L, and all external nodes have the same black depth.

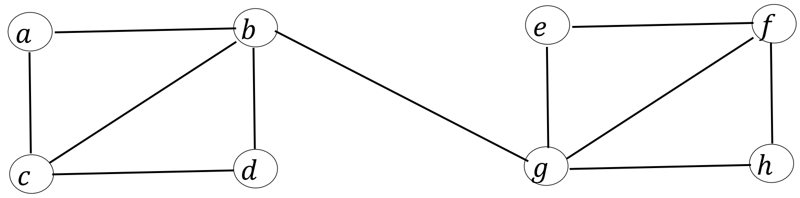
Marking Note – You may get a lot of students mapping a right-leaning red-black tree to a 2-3 tree first and then invoking the theorem we proved in class that all 2-3 trees map to a left-leaning red-black tree. This is okay.

2. [2 + 3 marks]
If you insert the elements of a permutation of n elements, in the order of the permutation, then inversions only occur when elements in the tree are bigger than the element being inserted. During an insertion, every time the current key k is less than the key in the current node, h, we add 1 + $size(h.right)$ to the inversion count. Thus, put() looks like the following (assuming variable invCount):

```
private Node put(Node h, int key) {
   if (h == null)
      return new Node(key, RED, 1);

   int cmp = key - h.key;
   if      (cmp < 0) {
      invCount += 1 + size(h.right);
      h.left  = put(h.left,  key);
      }
   else if (cmp > 0)
      h.right = put(h.right, key);
   else
      h.key   = key;

   if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
   if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
   if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
   h.size = size(h.left) + size(h.right) + 1;

   return h;
  }
```

3. [1 + 2 + 2 marks]

(a)



(b) There are 3 paths from a to b. For every one of those there is 1 path from b to g and for every one of those there are 3 paths from g to h. So, by the rule of products there are a total of $3 \times 1 \times 3 = 9$ paths from a to h.

(c) There is 3 path with length less than 5. They are

$$a, b, g, h$$
$$a, b, g, f, h$$
$$a, c, b, g, h$$

4. [5 marks]

Let G=(V,E) be an undirected graph, with no parallel edges or self-loops. Let |V|=n and |E|=m. For the induction, let m_i denote the number of edges for each n=i.

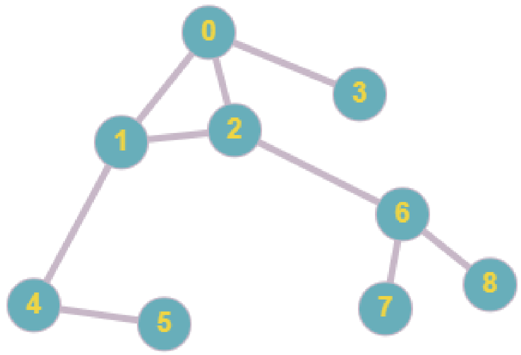
When n=1, the graph must have no edges, thus $2m_1 = 0$ and $n_2 - n = 1 - 1 = 0$ and the base case holds.

Let n=k and assume that $2m_k \leq k^2 - k$. Now, suppose n = k + 1 in a graph with m_{k+1}edges. Remove one vertex, say v, and all edges incident upon it. We are now left with a graph with k vertices and m_k edges. By induction then, $2m_k \leq k^2 - k$. In this graph, we know that the total degree for all the vertices is $2m_k$, thus if we add v and the its incident edges (at most k of them) back to the graph we have

$$\begin{aligned} &2m_{k+1} = 2m_k + 2\deg(v) \\ &\leq k^2 - k + 2k \\ &= k^2 + 2k + 1 - k - 1 \\ &= (k+1)^2 - (k+1) \end{aligned}$$

Therefore, by induction $2m \leq n^2 - n$ for all $n \geq 1$.

5. [2 + 1 + 2 marks]
Consider the graph *G* shown below:



(a) $2^9 = 512$. There are 9 edges each of which is either in the spanning subgraph or it isn't. All the vertices must be in the spanning subgraph.

(b) 4. The graph itself plus one for each edge in the cycle 0, 1, 2.

(c) $2^6 = 64$. Remove edges (0,1), (0,2), and (0,3) to isolate 0. The rest of the 6 edges are either in the subgraph or are not.