University of Washington

Department of Computer Science and Engineering

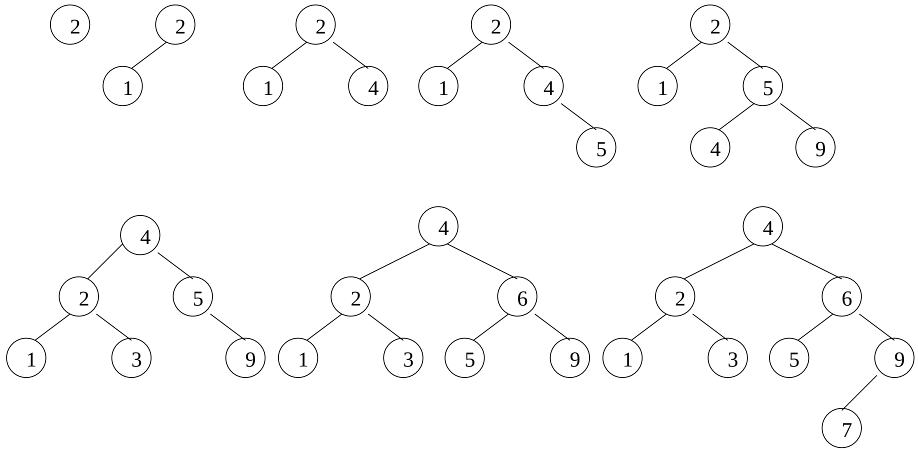
CSE 373

Problem Set 3 Solution

1. (1 point) Listing Nodes in Level-order

   The required structure is a queue. Listing nodes in level order is basically doing a "breadth-first search" of the tree, starting from the root.

2. (2 points) AVL Tree Insertion

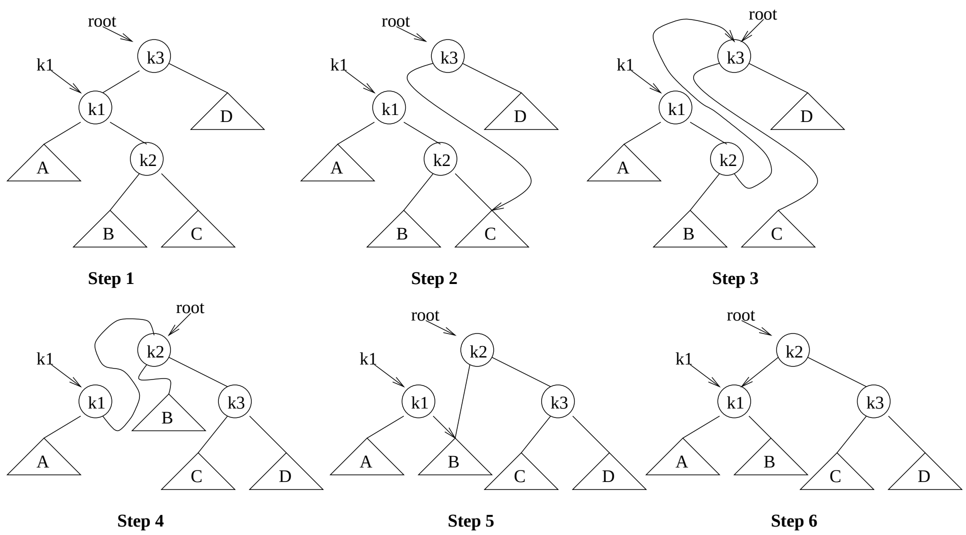


3. (5 points) Left-right Double Rotation [NOTE: The assignment actually was to do right-left, which is analogous. Sorry for the mix-up!]

```
Left_Right_Rotate(var NodePtr root)
{
  /* 1 */ NodePtr k1 = root->left;         // root is k3 in book

  /* 2 */ root->left = k1->right->right;  // Move C to left of k3
  /* 3 */ k1->right->right = root;        // Move k3 to right of k2
  /* 4 */ root = k1->right;               // Move k2 to the root
  /* 5 */ k1->right = root->left;         // Move B to right of k1
  /* 6 */ root->left = k1;                // Move k1 to left of k2
}
```

Here is a trace of the program running on a sample tree:



4. (12 points) Tree Counting Algorithms

   (a) Number of nodes in T.

```
int num_nodes(NodePtr p)
{
  if (p == NULL)
    return 0;
  else
    return 1 + num_nodes(p->left) + num_nodes(p->right);
}
```

   (b) Number of leaves in T.

```
int num_leaves(NodePtr p)
{
  if (p == NULL)
    return 0;
  else if (p->left == NULL && p->right == NULL)
    return 1;
  else
    return num_leaves(p->left) + num_leaves(p->right);
}
```

   (c) Number of full nodes in T.

```
int num_full(NodePtr p)
{
  if (p == NULL)
```

```
      return 0;
    else if (p->left == NULL)
      return num_full(p->right);
    else if (p->right == NULL)
      return num_full(p->left);
    else
      return 1 + num_full(p->left) + num_full(p->right);
  }
```

Note that all of these routines run in linear time, since they consider a given node exactly once. Another way to see this is that they are all variations of the basic in/pre/post order traversals.

5. (10 points) **Extra credit.** Tree similarity (solution outline).

   Deciding if two trees are similar can be done via a simple recursive procedure similar to those in Problem 4. Here is the pseudocode for the routine.

```
boolean similar(NodePtr p1, NodePtr p2)
{
  // If both are null, the trees are similar
  if ((p1 == NULL) && (p2 == NULL))
    return TRUE;

  // If one is null and the other isn't, they are not similar.
  else if ((p1 == NULL) || (p2 == NULL))
    return FALSE;

  // If we get here, neither is null so we must compare their children.

  // If the left children and the right children are each similar,
  // the trees are similar.
  return similar(t1->left, t2->left) && similar(t1->right, t2->right);
}
```

   If the two trees have size m and n respectively, this routine runs in $O(m+n)$ time.

   NOTE: the simplest way to represent a tree for input is to define it recursively. Under this definition, a tree is either empty and is represented by "()" or is represented by the pattern "(DATA TREE TREE)". For example, the rightmost tree on the top row of the problem 2 solution would be

```
(2 (1 () ()) (5 (4 () ()) (9 () ())))
```

   Given this representation, one can easily write a recursive C function to construct the tree.