These must be completed and shown to your lab TA either by the end of this lab, or at the start of your next lab. You may work in groups of up to two people.

1. Familiarize yourself with the Binary Search Tree (BST) data structure. Even if you have no experience with BST's, everything you need to know about them for lab 5 is explained or linked to below.

   This video covers binary tree basics: `https://www.youtube.com/watch?v=sf_9w653xdE`.
   Some terminology:

   - **root**: the top node in a tree (the only node with no parent)
   - **parent (of child)**: the immediate ancestor of child (where parent-child path exists)
   - **siblings**: nodes with the same parent
   - **descendant (of child)**: a node in the left or right subtrees of child
   - **ancestor (of child)**: a node in the path from root to parent (where parent-child path exists)
   - **leaf**: a node with no children.(also called an external node; all others nodes are internal)
   - **edge**: the connection between parent and child nodes (don't worry about this one; in bst.cc there are no edges)
   - **path**: a sequence of nodes and edges connecting a node with a descendant of the node
   - **path-length**: given a path, the path-length is the number of edges in the path, which is equal to (number of nodes) - 1

   First, a BST is a binary tree. That means it's either empty or it has one root node, and every node has at most two children. BST nodes will have at least three data members:

   - a key
   - a pointer to its left child (a value of NULL means there is no left child)
   - a pointer to its right child (a value of NULL means there is no right child)

   Search Tree Property: All the keys in the left subtree are less than the current node's key, and all the keys in the right subtree are greater than the current node's key.

2. Download the binary search tree code from the course web page under Lab 5. You can compile them using `make`. There over 40 failing tests which you must pass. You can run the tests with:

   ```
   ./bst
   ```

   You also have the ability to write your own testing/debugging code and run that instead. To do that, just supply your test keys as command line arguments:

   ```
   ./bst 5 3 2 1 6 8 4 7 9
   ```

   You can also add your own test code to runMain(). Use this to help you with debugging and testing!

   To fix the failing tests, implement the following functions in `bst.cpp`.

3. **(a)** Since this is a recursive structure (a binary tree is either empty, or it is a root node, together with at most two children, both of which are binary trees), recursive functions will work best. If you need a jump-start for this function, watch the first 5 minutes of `https://www.youtube.com/watch?v=sqVefIEttT0`.

   ```
   /**
    * Returns the number of nodes in the tree rooted at root.
    */
   int numNodes( Node* root );
   ```

(b) The algorithm for numLeaves() will be very similar to numNodes(), with a couple of important differences. For one, the base case will be different.

```
/**
 * Returns the number of leaves in the tree rooted at root.
 */
int numLeaves(Node* root);
```

(c) The height of any tree is defined as the greatest path-length from root-to-leaf (see page 1 for path-length definition). A tree with only one node has height = 0 and an empty tree has height = -1. The height of a node is defined as the greatest path-length from the node to a leaf. If you think of the node as the root of a subtree, the height of the node is the height of that subtree.

```
/**
 * Returns the height of node x.
 */
int height( Node* x );
```

(d) The depth of a node relative to an ancestor is the length of the path from the ancestor to the node.

```
/**
 * Returns the depth of node x in the tree rooted at root.
 */
int depth( Node* x , Node* root );
```

4. Three ways to traverse a tree (i.e. to visit all nodes in the tree) are pre-order, in-order, and post-order. They are described below, and also in this video `https://www.youtube.com/watch?v=xoU69C4lKlM`. The first minute explains all you really need to know, but it's instructive to watch him build the BST. To "visit" a node, you will use the `Visitor` variable `v`. You're not expected to know how it works (but feel free to check it out! Learning is fun!), but it's a reference to a `Visitor` object, and you can "visit" a node with:

   ```
   v.visit(rootNode, level);
   ```

   In-order traversal visits the nodes in ascending order: it firsts visits the left subtree, then the current node, then the right subtree. An analogy for in-order traversal is arithmetic notation (3+4); operator + is in-between the two operands. **(a)**

   ```
   /**
    * Traverse a tree rooted at rootNode in-order and use 'v' to visit each node.
    */
   void in_order( Node*& rootNode, int level, Visitor& v );
   ```

   (b) Pre-order traversal first visits the current node, then the left subtree, then the right subtree. An analogy for pre-order traversal is the way we might invoke an add() function, i.e. add(3,4). First there is the operator, then the two operands. The first node visited by a pre-order traversal will always be the root of the tree. (That fact can be useful if you're given the output of the three traversals and asked to reconstruct the tree itself.)

   ```
   /**
    * Traverse a tree rooted at rootNode pre-order and use 'v' to visit each node.
    */
   void pre_order( Node*& rootNode, int level, Visitor& v );
   ```

(c) Post-order traversal first visits the left subtree, then the right subtree, then the current node. An analogy for post-order traversal is **Reverse Polish Notation**. Some very early calculators and computers used this. You'd enter the first number in R0 (register zero) and push it onto the stack, then enter the second number in R0. The plus instruction would add whatever was on the top of the stack to whatever was in R0 (putting the result in R0, and decrementing the stack pointer). (Another fact that might be useful, if you're given some traversal outputs and asked to reconstruct the tree itself, is which node is always visited last by post-order traversal.)

```
/**
 * Traverse a tree rooted at rootNode post-order and use 'v' to visit each node.
 */
void post_order( Node*& rootNode, int level, Visitor& v );
```

5. When that is done, you can complete the missing portion of `delete_node` in `bst.cpp`. The easy cases are done: the node to be deleted is a leaf (case 1), or only has a left child (case 3), or only a right child (case 4). The difficult case, where the node has two children (case 2), we've thoughtfully left for you. Check out `https://www.youtube.com/watch?v=82cIlfCkCCw` to see a video of how this is done, and play around with inserting and removing nodes at `https://www.cs.usfca.edu/~galles/visualization/BST.html`. Replace the target node to be deleted with its predecessor, which is the target node's right-most descendent of its left subtree (i.e. the node in the BST that is just smaller than the target node).

```
/**
 * Deletes a node containing 'key' in the tree rooted at 'root'.
 */
bool delete_node(Node*& root, KType key);
```

6. Which functions would change if the Nodes were part of a binary tree that didn't have the search tree property (the invariant that requires `left < parent < right`)?

7. Be sure to show your work to your TA before you leave, or at the start of the next lab, or you will not receive credit for the lab!