**Binary search tree**

Binary search trees (BSTs) are very simple to understand. We start with a root node with value *x*, where the left subtree of *x* contains nodes with values *< x* and the right subtree contains nodes whose values are $\geq $ *x*. Each node follows the same rules with respect to nodes in their left and right subtrees. BSTs are of interest because they have operations which are favorably fast: insertion, look up, and deletion can all be done in *O*(*log n*) time. It is important

to note that the *O*(*log n*) times for these operations can only be attained if the BST is reasonably balanced; for a tree data structure with self-balancing properties see AVL tree. In the following examples you can assume, unless used as a parameter alias that *root* is a reference to the root node of the tree.



The class representing the Binary search tree is the following one

class Node {

 int data;

 Node \*left;

 Node \*right;

}

For the insertion, deletion and traversal operations we need to define the following methods of the class:

 Node() { data = -1; left = NULL; right = NULL; };

 void setKey(int key) { data = key; };

 void setLeft(Node\* Left) { left = Left; };

 void setRight(Node\* Right) { right = Right; };

 int getKey() { return data; }

 Node \*getLeft() { return left; }

 Node \*getRight() { return right; }

As mentioned previously insertion is an *O*(*log n*) operation provided that the tree is moderately balanced.

The insertion algorithm is split for a good reason. The first algorithm (non-recursive) checks a very core base case – whether or not the tree is empty. If the tree is empty then we simply create our root node and finish. In all other cases we invoke the recursive *addNode* algorithm which simply guides us to the first appropriate place in the tree to put *value*. Note that at each stage we perform a binary chop: we either choose to move into the left subtree or the

right by comparing the new value with that of the current node. For any totally ordered type, no value can simultaneously satisfy the conditions to place it in both subtrees.

void addNode(int key, Node \*leaf)

 {

 if (leaf->getKey() == -1)

 {

 leaf->setKey(key);

 }

 else

 {

 if (key <= leaf->getKey()) {

 if (leaf->getLeft() != NULL)

 addNode(key, leaf->getLeft());

 else {

 Node \*n = new Node();

 n->setKey(key);

 leaf->setLeft(n);

 }

 }

 else {

 if (leaf->getRight() != NULL)

 addNode(key, leaf->getRight());

 else {

 Node \*n = new Node();

 n->setKey(key);

 leaf->setRight(n);

 }

 }

 }

 }

**Deletion**

Removing a node from a BST is fairly straightforward, with four cases to consider:

1. the value to remove is a leaf node; or

2. the value to remove has a right subtree, but no left subtree; or

3. the value to remove has a left subtree, but no right subtree; or

4. the value to remove has both a left and right subtree in which case we promote the largest value in the left subtree.

Of course, in a BST a value may occur more than once. In such a case the first occurrence of that value in the BST will be removed.

