How to handle duplicates in Binary Search Tree?

In a Binary Search Tree (BST), all keys in left subtree of a key must be smaller and all keys in right subtree must be greater. So a Binary Search Tree by definition has distinct keys.

How to allow duplicates where every insertion inserts one more key with a value and every deletion deletes one occurrence?

A **Simple Solution** is to allow same keys on right side (we could also choose left side). For example consider insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree

```
          12
       /     \
     10       20
    /  \     /
   9   11   12
      /      \
    10       12
```

A **Better Solution** is to augment every tree node to store count together with regular fields like key, left and right pointers.
Insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree would create following.

```
          12(3)
       /        \
     10(2)      20(1)
    /    \
 9(1)   11(1)

Count of a key is shown in bracket
```

This approach has following advantages over above simple approach.

1) Height of tree is small irrespective of number of duplicates. Note that most of the BST operations (search, insert and delete) have time complexity as O(h) where h is height of BST. So if we are able to keep the height small, we get advantage of less number of key comparisons.

2) Search, Insert and Delete become easier to do. We can use same insert, search and delete algorithms with small modifications (see below code).

3) This approach is suited for self-balancing BSTs (AVL Tree, Red-Black Tree, etc) also. These trees involve rotations, and a rotation may violate BST property of simple solution as a same key can be in either left side or right side after rotation.

Below is C implementation of normal Binary Search Tree with count with every key. This code basically is taken from code for insert and delete in BST. The changes made for handling duplicates are highlighted, rest of the code is same.

```
// C program to implement basic operations (search, insert and delete)
// on a BST that handles duplicates by storing count with every node
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    int count;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    temp->count = 1;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d(%d) ", root->key, root->count);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    // If key already exists in BST, icnrement count and return
    if (key == node->key)
    {
       (node->count)++;
        return node;
    }

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the node with
   minimum key value found in that tree. Note that the entire
   tree does not need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

/* Given a binary search tree and a key, this function
   deletes a given key and returns root of modified tree */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the
    // root's key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key
    else
    {
        // If key is present more than once, simply decrement
        // count and return
        if (root->count > 1)
        {
           (root->count)--;
           return root;
        }

        // ElSE, delete the node

        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
             12(3)
          /        \
       10(2)      20(1)
       /   \
    9(1)  11(1)   */
    struct node *root = NULL;
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 9);
    root = insert(root, 11);
    root = insert(root, 10);
    root = insert(root, 12);
    root = insert(root, 12);

    printf("Inorder traversal of the given tree \n");
    inorder(root);

    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 12\n");
    root = deleteNode(root, 12);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 9\n");
    root = deleteNode(root, 9);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    return 0;
    return 0;
}
```

Output:

```
Inorder traversal of the given tree
9(1) 10(2) 11(1) 12(3) 20(1)
Delete 20
Inorder traversal of the modified tree
9(1) 10(2) 11(1) 12(3)
Delete 12
Inorder traversal of the modified tree
9(1) 10(2) 11(1) 12(2)
Delete 9
Inorder traversal of the modified tree
10(2) 11(1) 12(2)
```

We will soon be discussing AVL and Red Black Trees with duplicates allowed.