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;
}
```

Run on IDE

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.

This article is contributed by **Chirag**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

13 Comments Category: Binary Search Tree

Related Posts:

- Print Common Nodes in Two Binary Search Trees
- Construct all possible BSTs for keys 1 to N
- K'th smallest element in BST using O(1) Extra Space
- Count BST subtrees that lie in given range
- Count BST nodes that lie in a given range
- Data Structure for a single resource reservations
- Advantages of BST over Hash Table
- K'th Largest Element in BST when modification to BST is not allowed

0 Average Difficulty : **0/5.0**
No votes yet.

(Login to Rate)

Like Share 12 people like this. Sign Up to see what your friends like.

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

13 Comments GeeksforGeeks 1 Login

Recommend Share Sort by Newest

Join the discussion…

Udit Agrawal • 4 months ago

can't we replace the part of delete function in which there are both children with this

```
node *right = root->right;
node *temp = minValueNode(root->right);
temp->left = root->left;
root->left = root->right = NULL;
free(root);
return right;
```

• Reply • Share ›

DS+Algo • 5 months ago

Bug detected in deletion: Check the output here: http://ideone.com/GrVRNg
I corrected. See here: http://ideone.com/1D2Zu4
See lines 122-124

The bug is:
When node to be deleted has both children, key of inorder successor is copied and inorder successor is deleted but if inorder successor has count>1 then its count is decremented but it's not deleted, so we have now two nodes with same key.

Solution:
inorder successor's count should also be copied to the node to be deleted and make inorder successor's count=1 so that when it is tried to delete, it delete this node instead of decrementing count

10 • Reply • Share ›

codingfreak DS+Algo • 3 months ago

we can slightly modify the delete function when nodes to be deleted have 2 children

```
node *temp = minValueNode(root->right);
root->data = temp->data;
root->count = temp->count;
temp->count = root->count;
root->right = deleteNode(root->right,temp->data);
```

• Reply • Share ›

DS+Algo ➦ codingfreak • 2 months ago

temp->count=1 is needed.

• Reply • Share ›

codingfreak ➦ DS+Algo • 2 months ago

This statement should take care of it
temp->count = root->count

• Reply • Share ›

DS+Algo ➦ codingfreak • 2 months ago

you have already done root->count=temp->count;
after that, temp->count=root->count makes no sense because both variables will now contain same value.

• Reply • Share ›

codingfreak ➦ DS+Algo • 2 months ago

Thanks for correcting...

• Reply • Share ›

#InnerPeace ➦ DS+Algo • 4 months ago

Good Job Man.

@Think Loud(Noso)

• Reply • Share ›

Gaurav Arora ➦ DS+Algo • 4 months ago

Exactly why I came to the comments section.

2 • Reply • Share ›

vergil ➦ DS+Algo • 5 months ago

i was thinking the same!...good job!

• Reply • Share ›

vetiarvind • 5 months ago

I'm confused. It says "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"
Then how is the following possible:"A Simple Solution is to allow same keys on right side (we could also choose left side)"

Aren't you contradicting the 1s't quote in the 2nd one? Shouldn't the first line say " all keys in left subtree must be smaller or equal"

• Reply • Share ›

DS+Algo ➦ vetiarvind • 5 months ago

A BST assumes that all keys are distinct hence the first statement is true. In this article we are trying to develop BST such that it can work with duplicates too.

• Reply • Share ›

Jeffery yuan • 5 months ago

Seems there is flaw(bad smell) in the delete function:
it would change the current node to its inorder successor, and count 1. then decrement the count of the inorder successor.
This would let same key(the inorder successor) two places, later it may populate to more places.
It should replace the current node with inorder successor: copy value, and count. Then delete the inorder successor by calling deleteNode (root->right, temp->key, temp->count);

• Reply • Share ›