Hash Tables

a

```
bool is_ht(ht H)
{
    if (H == NULL)
        return false;
    if (!(H->m > 0))
        return false;
    if (!(H->n >= 0))
        return false;
    //@assert H->m == \length(H->table);
    int nodecount = 0;
    for (int i = 0; i < H->m; i++)
    {
        // set p equal to a pointer to first node
        // of chain i in table, if any
        chain *p = H->table[i];
        while ()
        {
            elem e = p->data;
            if ((e == NULL) || (abs(hash(elem_key(e))) % H->m != i))
                return false;
            nodecount++;
            if (nodecount > H->n)
                return false;
            p = p->next;
        }
    }
    if (nodecount != H->n)
        return false;
    return true;
}
```

b

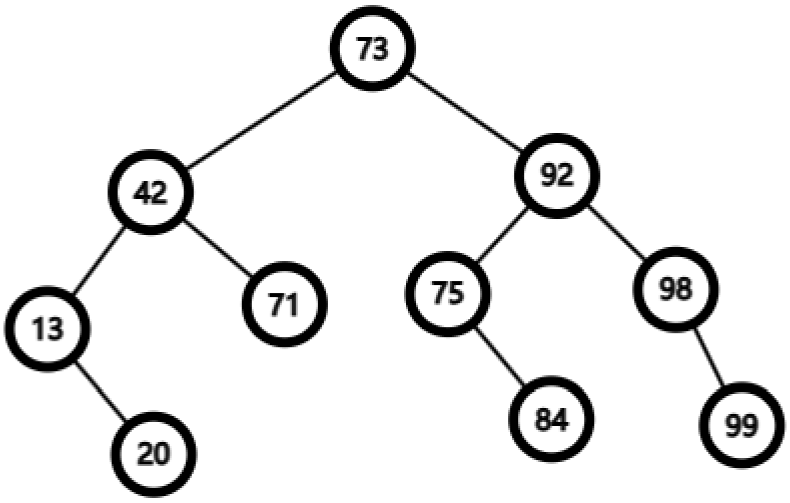
```
/*@ensures \result == NULL
            || key_equal(k, elem_key(\result))
*/
```

Binary Search Tree

a

75 92 99 13 84 42 71 98 73 20

13 20 42 71 73 75 84 92 98 99



b

Assume that $f[n]$ refers to the numbers of different binary tree using n nodes.Then consider for specific n, what index can be the root. So we can get the equation:

$$f[n] = \sum_{i=0}^{n-1}f[i]*f[n-i-1], f[0] = 1$$

So we can get $f[0]=1, f[1]=1, f[2]=2, f[3]=5, f[4]=14, f[5]=40$. For the following keys:73, 28, 52,-9, 104, we can get $f[5]=40$ different trees.

c

```
int tree_height(tree *T)
//@requires is_ordered(T, NULL,NULL);
{
    int max_deep = 1;
    if(T->left != NULL)max_deep = max(max_deep(tree_height(T->left)));
    if(T->right != NULL)max_deep = max(max_deep(tree_height(T->right)));
    return max_deep;
}
int bst_height(bst B)
//@requires is_bst(B);
//@ensures is_bst(B);
{
    return tree_height(B->root);
}
```

d

```
tree *tree_delete(tree *T, key k)
{
    if (T == NULL)
    { // key is not in thetree
        return T;
    }
    if (key_compare(k, elem_key(T->data)) < 0)
    {
        T->left = tree_delete(T->left, k);
        return T;
    }
    else if (key_compare(k, elem_key(T->data)) > 0)
    {
        T->right = tree_delete(T->right, k);
        return T;
    }
    else
    {                         // keyisin currenttreenode T
        if (T->left == NULL) // node hasonly right child
            return T->right;
        else if(T->right == NULL) // node hasonlyleft child
            return T->left;
        else
        {
            // Node to be deleted has two children
            if (T->left->right == NULL)
            {
                // Replace the data in T with the data
                // in the left child.
                T->data = T->left->data;
                // Replace the left child with its left child.
                T->left = T->left->left;
                return T;
            }
            else
            {
                // Search for the largest child in the
                // left subtree of T and replace the data
                // in node T with this data after removing
                // the largest child in the left subtree.
                T->data = largest_child(T->left);
                return T;
            }
        }
    }
}
elem largest_child(tree *T)
//@requires T != NULL && T->right != NULL;
{
    if (T->right->right == NULL)
    {
        elem e = T->right->data;
        T->right = NULL;
        return e;
    }
    return largest_child(T->right);
}
```